Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Prof. Dr. Ulrich Eisenecker

Thema

Reduktion von Quellcoderedundanz als Motivator der Evolution von Programmiersprachen am Beispiel von Java 8

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science - Wirtschaftsinformatik

vorgelegt von: Triebel, Anna Juliane
Prüfungsnummer: 15563
Matrikelnummer: 3693223
Email-Adresse: anna.j.triebel@gmail.com
Telefonnummer: +49 1573 1818 482
Anschrift: Lützner Straße 134
04179 Leipzig

Leipzig, den 9. April 2018

Abstract

Ist die Reduktion von Quellcoderedundanz ein Motivator für die Evolution von Programmiersprachen? Das ist die Ausgangsfrage der Untersuchung, die exemplarisch an Sprachfeatures von Java 8 beleuchtet wird. Code Clones und Boilerplate Code werden als Formen von Quellcoderedundanz aufgefasst, beschrieben und definiert. Quellcoderedundanz wird als das Verhältnis der Komplexität des Ausdrucks und der durch diesen transportierte Information definiert und operationalisiert. Zur Messung der Änderung der Quellcoderedundanz durch Java 8 werden Codesegmente von Java 7 auf Java 8 migriert. Bei konstantem Informationsgehalt wird die Ausdruckskomplexität durch Maße der statischen Codeanalyse verglichen. Die Untersuchung zeigt für alle betrachteten Sprachfeatures eine Abnahme der Quellcoderedundanz, die aus einer Reduktion von Boilerplate Code oder dem Wegfall von Code Clones resultiert. Die Ergebnisse deuten darauf hin, dass die Reduktion von Quellcoderedundanz für die mit Java 8 in die Sprache eingeführten Neuerungen zumindest eine notwendige Eigenschaft ist. Um im Ökosystem der Programmiersprachen weiter bestehen zu können, müssen sich Sprachen weiterentwickeln, da ihr technologisches Umfeld stets im Wandel ist. Um seinen Nutzern die Möglichkeit zu geben, qualitativ hochwertigen Quellcode zu verfassen, müssen Sprachmittel zur Verfügung gestellt werden, die eine elegante Ausdrucksform komplexer Sachverhalte erlauben. Eine geringe Quellcoderedundanz kann also als Qualitätsmerkmal für Quellcode gelten und deren Ermöglichung als Evolutionsvorteil für Programmiersprachen angesehen werden.

Schlüsselwörter

Redundanz, Code Clone, Boilerplate Code, Java 8, Programmiersprache, Codequalität

Inhaltsverzeichnis

Abkürzungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

1 Einleitung

Der vorliegende Text soll einen Beitrag zur Klärung der Frage leisten, ob Redundanzreduktion als treibender Faktor für die Entwicklung von Programmiersprachen angesehen werden kann. Zunächst wird ein Begriffssystem erarbeitet, das in einem zweiten Schritt dazu verwendet wird, die oben genannte These an ausgewählten Beispielen empirisch zu prüfen. Dazu wird *Quellcoderedundanz* als Oberbegriff von *Code Clones* (CC) und *Boilerplate Code* (BC) beschrieben und die Neuerungen der Programmiersprache Java in Version 8 werden daraufhin untersucht, inwieweit sie zu weniger redundantem Quellcode beitragen können.

Aktuelle Forschung und zahlreiche Veröffentlichungen zu Code Clones, deren Auffindung, Typisierung, Beseitigung und Vermeidung, sowie Tools, die Klone finden, nachverfolgen oder refaktorieren zeigen die Praxisrelevanz der Thematik auf. Das Auftreten von Code Clones wird dabei allermeist mit der Gefahr von minderer Codequalität und letztlich hohen Wartungs- und Instandhaltungskosten für Software in Verbindung gebracht. Obwohl zu Boilerplate Code starke Parallelen bestehen, werden beide Phänomene in der Literatur selten zusammen erwähnt. Boilerplate Code wird häufig in Zusammenhang mit Sprach- und *application programming interface* (API)-Design diskutiert. Drauf aufbauend soll eine Untersuchung der Möglichkeit systematischer Vermeidung von Quellcoderedundanz durch die Evolution von Programmiersprachen folgen.

Kapitel 2 soll die theoretische Grundlage für die im Kpitel 3 folgende empirische Betrachtung leisten. Unerwünschte Redundanz in Quellcode soll an zwei Phänomenen expliziert werden. Erstens werden Code Clones, bei der Programmierung entstehende Dopplungen von Quellcode, erläutert. Weiter wird das Auftreten von Boilerplate Code als von der Programmiersprache aufgezwungene Wiederholung von Quellcodesegmenten betrachtet. Beide Erscheinungsformen von Redundanz sollen definiert, abgegrenzt und ihre Gemeinsamkeiten im Oberbegriff Quellcoderedundanz abstrahiert werden. Zur konkreten Prüfung der Ausgangsthese werden im zweiten Teil der Arbeit markante, mit Java 8 eingeführte Sprachmittel auf ihr redundanzvermeidendes Potential untersucht. Einerseits wird dabei die Art der Redundanzreduktion betrachtet und weiter deren Bedeutung im Kontext der Entwicklung von Java beleuchtet. Bestätigt sich die Ausgangsthese, so kann vermutet werden, dass von zwei möglichen Lösungen die weniger redundante bessere Codequalität liefert. Auf dieser theoretischen Basis wären dann nicht nur Handreichungen für die Programmierpraxis, sondern auch Prognosen für die Weiterentwicklung von Programmierkonzepten und -sprachen möglich.

2 Programmiersprachen und Quellcoderedundanz

„Clones can only be grown so quickly if you want them mentally stable enough to trust with your warships.“ Luke Skywalker [o.V. 2017]

Der erste Teil der Arbeit soll die theoretische Grundlage für die im zweiten Teil folgende empirische Betrachtung leisten. Unerwünschte Redundanz im Quellcode soll an zwei Phänomenen expliziert werden. Erstens werden Code Clones, bei der Programmierung entstehende Dopplungen von Quellcode, erläutert. Weiter wird das Auftreten von Boilerplate Code als von der Programmiersprache aufgezwungene Wiederholung von Quellcodesegmenten betrachtet. Beide Erscheinungen von Redundanz sollen definiert, verglichen und ihre Gemeinsamkeiten im Oberbegriff *Quellcoderedundanz* abstrahiert werden. Die Möglichkeiten, wenig redundanten Quellcode zu schreiben, hängt zum Einen von der Beschaffenheit der Programmiersprache ab, andererseits vom Einsatz der von ihr bereitgestellten Sprachmittel. Das Wie der Realisierung von Konzepten in einer Sprache hat Einfluss auf die potentielle Redundanz des Quellcodes.

2.1 Code Clones

Als CCs oder *Duplicate Code* werden zwei oder mehr Segmente in demselben Softwaresystem bezeichnet, die syntaktisch oder semantisch gleich oder ähnlich sind (vgl. [Bellon et al. 2007, S. 471]). So oder ähnlich werden Code Clones oft umschrieben. Auch wenn das Phänomen bekannt ist, liegt bisher keine einheitliche Definition vor und in vielen Veröffentlichungen wird gar keine solche Beschreibung geliefert, sondern eine Definition erfolgt allenfalls implizit durch Operationalisierung im Forschungsvorhaben (vgl. [Higo et al. 2008, S. 436]). Taxonomien von CCs gibt es auf verschiedenen Dimensionen, wie deren Entstehung, dem Grad ihrer Ähnlichkeit oder der Art ihrer Auffindung (vgl. [Kumar/Singh 2015]).

Die Sichtweise auf Duplikate in Quellcode unterliegt in ihrer Geschichte einem Deutungswandel von strenger Verteufelung hin zu der Sichtweise, dass nicht alle Klone unmittelbar Unheil bedeuten. Die Praxis hat gezeigt, dass eine strenge Vermeidung nicht möglich ist und daher gibt es einen Trend zur Akzeptanz von Duplikaten, der den Fokus mehr auf Klon-Management legt als auf ein strenges Verbot. Denn Dopplungen werden teilweise bewusst erzeugt, wenn in Abwägung gegen andere Risikofaktoren kontrolliertes Klonen eine akzeptable Problemlösung bietet. Kapser und Godfrey weisen darauf hin, dass Dopplungen in bestimmten Szenarien positive Effekte haben[1]. Sie

[1] Beispielsweise werden beim *Forking* ähnliche Codeabschnitte für verschiedene Hardware, Betriebssysteme oder als von Produktivcode abgesondertes Experimentierfeld parallel in verschiedenen Varia-

können als *gute Clones* bezeichnet werden und entstehen als bewusste Designentscheidung (vgl. [Kapser/Godfrey 2008]). Unbeabsichtigt oder aus den falschen Gründen entstandene *böse Clones* gelten weiterhin als *Code Smells*, das heißt als Indikatoren für Designschwächen und potentielle Quelle für Probleme (vgl. [Fowler/Beck 1999]). Hier soll der Fokus auf der letzteren Art, ihrer Auffindung (vgl. [Smith/Horwitz 2009; Bellon et al. 2007]), Beseitigung (vgl. [Speicher/Bremm 2013; Mazinanian et al. 2016; Arefin/ Khatchadourian 2015]) und letztendlich ihrer Vermeidung liegen. Diese Schwerpunktsetzung bildet die Konzentration der Forschungsaktivitäten auf dem Gebiet ab und soll einen Überblick über zu dieser Arbeit in Beziehung stehende Veröffentlichungen geben. Ist in solchen Studien von CCs die Rede, sind damit allermeist unerwünschte Klone gemeint. Nicht nur bergen sie wirtschaftliche Risiken, sie werden aufgrund ihrer logischen Überflüssigkeit oft als unästhetisch wahrgenommen (vgl. [Martin 2009, S. 48]). Mehr Quellcode bedeutet größeren Einarbeitungsaufwand für die Entwickler und schlichtweg mehr Material, das es zu warten gilt. Zudem wird Verständlichkeit gemindert, wenn dieselbe Funktionalität durch unterschiedliche Konstrukte bereitgestellt wird. Muss eine mehrfach implementierte Funktionalität geändert werden, besteht das Risiko, dass nicht alle betreffenden Codesegmente gefunden und angepasst werden und sich so Fehler einschleichen, deren Ursache nur schwer auszumachen ist. Die Schwierigkeit liegt darin, dass syntaktisch sehr ähnliche Zeichenketten semantisch sehr unterschiedlich sein können und syntaktisch verschiedene Codesegmente eventuell dieselbe Semantik ausdrücken (vgl. [Smith/Horwitz 2009, S. 3]).

In zahlreichen Formulierungen wird von namhaften Größen des Software Engineerings vor der Entstehung von CCs gewarnt oder zu deren Vermeidung aufgerufen. Das Prinzip *„Don't repeat yourself“* [Hunt/Thomas 2000, S. 27] (DRY) nach Andrew Hunt und David Thomas ist die vielleicht prominenteste Formulierung dieser Art. In positiver Wendung lautet sie: *„Every piece of knowledge must have a single, unambiguous, authoritative representation within a system“*[2] [Hunt/Thomas 2000, S. 27]. Zur Umsetzung des DRY-Prinzips wird Wiederverwendung angeraten (vgl. [Hunt/Thomas 2000, S. 33]), die oft durch Abstraktion umgesetzt werden kann. Kent Beck nennt *„Once and only once“* [Beck 1999, S. 71] ein Kernprinzip des *Extreme Programming*, Robert C. Martin zeigt „*Duplication*“ als *Code Smell* auf (vgl. [Martin 2009, S. 289]). Moderne Forschung fokussiert häufig eine automatisierte Lösung der Problematik. C. K. Roy et al. liefern eine Definition, die auf der ebenfalls von ihnen vorgestellten Typisierung aufbaut. Danach ist ein Klon ein Codefragment $CF1$, das nach einer gege-

tionen betrieben, die große Ähnlichkeiten aufweisen (vgl. [Kapser/Godfrey 2008, S. 651ff]).

[2] Jedes Stück Wissen muss eine einzige, eindeutige, verbindliche Repräsentation innerhalb eines Systems haben (Übers. d. A.).

benen Definition von Ähnlichkeit, zu einem anderen Codefragment $CF2$ ähnlich ist. Ein Fragment macht dabei eine oder mehrere Zeilen Quellcode aus. Ähnlichkeit besteht, wenn $f(CF1) = f(CF2)$ gilt, wobei f eine Ähnlichkeitsfunktion ist, die sich auf unterschiedliche Typen von Clones beziehen kann (vgl. [Roy et al. 2009, S. 471f]). Die Unterscheidung in syntaktische, also bezüglich der Ausdrucksweise und semantische, in ihrer Bedeutung ähnliche Klone lässt sich noch feiner untergliedern, wobei die interne Ähnlichkeit der Klonpaare oder -gruppen betrachtet wird (vgl. [Svajlenko/Roy 2014]). Von Typ-1-Klonen ist die Rede, wenn sich die Codefragmente lediglich in der Formatierung, Leerzeichen und Kommentierungen unterscheiden. Enthalten die Fragmente verschiedene Bezeichner, Literale oder Typen, liegen Klone vom Typ 2 vor. Variieren außerdem die im Code enthaltenen Anweisungen, spricht man von Klonen des Typs 3. Alle bisher beschriebenen Typen unterscheiden sich lediglich hinsichtlich ihrer Syntax. Funktionale Klone (Typ 4) führen dieselbe Programmlogik aus, sind jedoch semantisch verschieden implementiert (vgl. [Bellon et al. 2007, S. 472], [Roy et al. 2009, S. 472]). Diese Definition entspricht einer Explizitmachung der für die Klonerkennung zugrunde gelegte Taxonomie von Abstufungen der Ähnlichkeit. Eine saubere Abgrenzung zu Boilerplate Code liefert sie jedoch nicht, was in Kapitel 2 noch weiter diskutiert werden wird. Zunächst sollen die Phänomenologie von und der Forschungsstand zu Code Clones genauer betrachtet werden.

Obwohl die mit ihnen einhergehenden Risiken bekannt sind, entstehen Klone oft unabsichtlich oder aufgrund von Architektur- und Designschwächen im Code sowie schlechter Organisation und Kommunikation in Entwicklerteams (vgl. [Hunt/Thomas 2000, S. 33]). Diese weichen Faktoren als Ursache für Duplikationen in Quellcode können wie im Folgenden beschrieben klassifiziert werden. Unabsichtliche Duplikation entsteht oft aufgrund von Designfehlern, wenn Entwickler gar nicht merken, dass sie Informationen duplizieren. Geschieht dies durch unterschiedliche Entwickler, kann Interentwickler-Duplikation durch ein klares Design, einen kompetenten technischen Projektleiter und klare Verteilung der Verantwortlichkeiten innerhalb des Designs entgegengewirkt werden (vgl. [Hunt/Thomas 2000, S. 30ff]). Entwickler duplizieren, im Falle ungeduldiger Duplikation bewusst, weil es ihnen einfacher erscheint, bestehenden Code zu kopieren und anzupassen, anstatt die Gemeinsamkeiten der betreffenden Codeabschnitte zu abstrahieren oder gar das Design des Softwaresystems zu erneuern. Oft wird aufgrund von Zeitdruck in Projekten eine, für den Moment, schnellere aber unsaubere Lösung gewählt, die für weitere Änderungen in der Zukunft allerdings sehr viel mehr Aufwand erfordert (vgl. [Hunt/Thomas 2000, S. 32]). Man spricht dann, nach einer von Ward Cunningham erdachten Metapher, von technischer Schuld (vgl. [Cunningham 2018]). Für die hier vorliegende Fragestellung besonders interessant ist die

aufgezwungene Duplikation, die entsteht, wenn Entwickler den Eindruck haben, dass ihr technologisches Umfeld die Duplikation erzwingt. Dem Entwickler ist also durchaus bewusst, dass gerade Redundanz entsteht, die Vermeidung dieser liegt aber häufig nicht in seinem Einflussbereich. werden diese Dopplungen durch die jeweilige Sprache erzwungen werden, sind sie nur schwer zu vermeiden. Deshalb können viele Entwicklungsumgebungen (IDEs) solche aufgezwungenen Dopplungen generieren (vgl. [Hunt/ Thomas 2000, S. 28f]).

Das Auffinden von Klonen ist ein aktuelles Forschungsfeld, das verschiedene Techniken zur automatisierten Erkennung von Klonen hervor gebracht hat. Diese unterscheiden sich in Mächtigkeit und Aufwand. Je weniger ähnlich die Klone sind, desto schwieriger ist ihre Identifikation und dementsprechend müssen zu ihrer Auffindung aufwändige Verfahren zum Einsatz kommen. Die automatische Erkennung von semantischen Klonen birgt dabei die besondere Schwierigkeit, dass Bedeutung von Maschinen (noch) nicht eigenständig erfasst werden kann, was sich wohl erst mit dem Fortschreiten der Forschung zu künstlicher Intelligenz nennenswert weiter entwickeln wird. Bereits eine saubere Erkennung nicht identischer, syntaktischer Klone gestaltet sich als Herausforderung. Die Zielsetzung für alle Tools ist es, möglichst viele der tatsächlich im Code vorhandenen Klone zu finden und möglichst keine Codesegmente als Klone zu identifizieren, die gar keine sind. Ist ein Detection-Tool beziehungsweise der darin implementierte Erkennungsalgorithmus in der Lage, auch sehr unähnliche Klone zu erkennen, kommt es häufig zu *false positives*.

Exakte Klonerkennung ist ein bisher ungelöstes Problem. Dennoch soll ein kurzer Überblick über bestehende Ansätze zum Auffinden von Klonen vorgestellt werden. Textuelle Strategien, die auf dem Vergleich von Zeichenketten basieren, können Klone einschließlich Typ 3 identifizieren. Das Programm zum automatischen Auffinden von CCs namens *Dup* nutzt eine solche Technik (vgl. [Baker 1995]). Tokenbasierte Erkennungstechniken unterteilen den Quellcode in Tokens, die den lexikalischen Regeln der jeweiligen Programmiersprache entsprechen (vgl. [Baker 1993]) und ersetzen zudem Bezeichner durch Platzhalter, wie in *CCFinder* umgesetzt (vgl. [Kamiya et al. 2002]). Syntaxbasierte Ansätze parsen einen Syntaxbaum oder abstrakten Syntaxbaum, dessen Subbäume auf Ähnlichkeit geprüft werden. In *DECKHARD* ist eine solche baumbasierte Klon-Suche implementiert (vgl. [Jiang et al. 2007]). Die bisher vorgestellten Techniken kommen mit einer Komplexität von $O(n)$ aus. Sensibel für semantische Ähnlichkeit sind Techniken, die Kontroll- und Datenflüsse untersuchen. Sie sind mit $O(n^3)$ ungleich aufwändiger. *Duplex* nutzt diesen Ansatz (vgl. [Krinke 2001]). Dazu wird ein Programmabhängigkeitsgraph (PDG) aus dem Quellcode generiert, dessen Knoten die Statements und Zustände eines Programms abbilden. Die Kanten des Graphs repräsen-

tieren Kontroll- und Datenabhängigkeiten. Eine Isomorphie der Subgraphen des PDG weist dann auf das Vorhandensein von semantischen Klonen hin (vgl. [Bellon et al. 2007, S. 473]). Vergleichende Studien über die Genauigkeit der bisher hauptsächlich im akademischen Bereich entstandenen Clone-Detection-Tools zeigen, dass es für das Aufspüren möglichst vieler im Testcode enthaltenen CCs einer Kombination mehrerer Strategien bedarf. Weiter ist eine menschliche Gegenkontrolle der identifizierten Klon-Kandidaten vonnöten, da es häufig zu *false positives* kommt (vgl. [Rattan et al. 2013; Roy/Cordy 2008; Bellon et al. 2007]).

Wurden nun CCs identifiziert, schließt sich die Frage nach dem Umgang mit ihnen an. Soll ein Klon behalten oder eliminiert werden, gibt es verschiedene Strategien, das Risiko, dass ein Klon zu einem Bug führt, zu minimieren. Wird ein Klon als gut bewertet oder kann er nicht beseitigt werden, da er aus aufgezwungener Duplikation resultiert oder wird die Entscheidung getroffen, die mit der Tolerierung von Dopplungen einhergehende technische Schuld zunächst aufzunehmen, können Maßnahmen ergriffen werden, um bestehende Klone zu managen. Für so akzeptierte Klone stehen verschiedene Strategien, wie Dokumentation, Tracking, Linked Editing oder Generierung zu Verfügung. Zunächst können Klone, werden sie gefunden, durch Dokumentation als solche kenntlich gemacht werden, um einen Überblick über ihr Ausmaß zu erlangen. Geschieht dies händisch, kann es viele Personenstunden in Anspruch nehmen, wobei gerade in großen Softwaresystemen Vollständigkeit und Aktualität der Dokumentation nur schwer gewährleistet werden kann. Hier kann werkzeuggestütztes Nachverfolgen von CCs durch meist in die Entwicklungsumgebung integrierte Funktionalität genutzt werden. Dabei wird im Fall von Copy and Paste im Quellcode der neu entstandene Klone automatisch registriert (vgl. [Higo et al. 2013; Duala-Ekoko/Robillard 2010; Weckerle 2008]). Zu wissen, wo sich wie viele Klone befinden, ist ein großer Vorteil. Steht aber in einer großen Klongruppe eine Änderung an, bedeutet dies, dass für eine geänderte Funktionalität sehr viele Stellen im Quellcode analog angepasst werden müssen, was mit hohem Aufwand einhergehen kann. Für diese Problematik liefern einige der Tracking-Tools eine weitere Funktionalität, das Linked Editing wobei im Fall der Änderung eines Klons derselben Gruppe zugehörige Klone automatisch mit angepasst werden und so für korrekte Änderungssynchronisation sorgen (vgl. [Hou et al. 2009; Jablonski/Hou 2007; Toomim et al. 2004]). Werden die beschriebenen Techniken eingesetzt, ist es möglich, eine informierte Entscheidung über den Umgang mit CCs und der mit diesen einhergehenden technischen Schuld zu treffen. Da Quellcode zumeist nicht statisch ist, sondern ständiger Veränderung und Erneuerung unterliegt, muss die Frage nach dem Verbleib oder der Eliminierung von Klonen wiederholt gestellt werden. Wächst beispielsweise eine Klongruppe stark an oder muss sie sehr häufig angepasst

werden, kann der Abbau technischer Schuld durch Eliminierung der Klone mittel- oder langfristig eine kluge Entscheidung sein.

Durch *Refaktorierung*, also die Restrukturierung von Quellcode unter Beibehaltung der Funktionalität (vgl. [Fowler/Beck 1999; Arefin/Khatchadourian 2015; Gallardo 2003; Balzert 2011]), können bestimmte CCs eliminiert werden. Dabei kann die Konstanthaltung der Programmlogik eine Herausforderung darstellen, wenn nicht alle Klone der betreffenden Gruppe bekannt sind, sodass unbemerkt Inkonsistenzen entstehen. Für Klon-Refaktorierung werden, in Sprachen mit objektorientiertem Anteil wie Java, oft Konzepte wie Abstraktion und Vererbung genutzt, da durch sie die den Klonen gemeinsame Anteile einmal implementiert, jedoch beliebig wiederverwendet werden können. Auch in dieser Phase im Umgang mit Klonen helfen viele Entwicklungsumgebungen bei der Refaktorierung, zum Beispiel durch Extraktion der Funktionalität in eine höhere Abstraktionsebene (vgl. [Gallardo 2003]). Einen noch höheren Automatisierungsgrad bieten Werkzeuge wie *Aries*, die zuerst den Quellcode auf mögliche Code-Smells analysieren und im Anschluss Vorschläge für deren Behebung anbieten. Im Bezug auf Klone sind das Extrahieren in Methoden, Klassen, Interfaces, Module oder Superklassen, sowie das Erstellen einer neuen Methode, die den doppelten Code kapselt und die Ersetzung der Klone durch Methodenaufrufe (vgl. [Kumar/Chanakya 2014, S. 3289]). Auf Basis der Erkennungstechnik von CCFinder werden aus allen gefundenen Klonen diejenigen identifiziert, die für Refaktorierungen in Frage kommen. Auf diese werden die von Fowler vorgeschlagenen Techniken zur Zusammenführung redundanter Codesegmente ausgewählt (vgl. [Higo et al. 2008, S. 444]). Dafür ist die verhältnismäßige Position der Klone innerhalb der Klassenhierarchie maßgeblich sowie deren Grad der Kopplung an den sie umgebenden Code, der durch die Anzahl von referenzierten und zugewiesenen Variablen in quantitativen Maßen dargestellt wird (vgl. [Higo et al. 2008, S. 442]).

Die unnötige Komplexität redundanten Quellcodes erzeugt zusätzliche Arbeit, Risiken und höhere Instandhaltungs- und Änderungskosten der Software (vgl. [Kumar/Singh 2015]). Dagegen sind andere Qualitätsmerkmale von Quellcode abzuwägen, wie Lesbarkeit, Lokalität, Wiedererkennungswert oder absehbare Auseinanderentwicklung der Gemeinsamkeiten in der Zukunft. Clone-Forschung hat sich im Laufe ihrer Geschichte einem Wandel unterzogen. Zunächst wurde das Phänomen der Code Clones bemerkt und bezeichnet, wobei mit dem sehr strengen *„Once and only once“* eine rein negative Konnotation vorherrschte. Diesem strengen Anspruch konnte in der Praxis nicht nachgekommen werden, da mit Wachstum und Alterung von Quellcode allein der Überblick über existierende Klone einer Mammutaufgabe gleicht. Daraufhin folgte ein Trend zur Automatisierung solcher Aufgaben, der heute noch anhält. Die Sichtweise auf

Klone ist jedoch im Wandel, sodass Duplikate nicht mehr als per se schlecht angesehen werden, sondern die Frage nach einem sinnvollen Umgang mit Duplikaten gestellt und der Fokus mehr auf eine informierte Risikoabschätzung gelegt wird.

2.2 Boilerplate Code

Boilerplate Code bezeichnet von einer Programmiersprache direkt oder indirekt vorgegebene syntaktische Konstrukte, die häufig sperrig sind und bei der Programmierung oft wiederholt werden müssen. Entweder handelt es sich um Konstrukte, die zur Sprachsyntax selbst gehören oder um welche, die von Softwareentwicklern geschaffen werden, um Konstrukte zu bilden, die die Sprache selbst nicht bereit stellt. Sind diese standardisierten Textbausteine sehr groß, wird der Quellcode aufgebläht, ohne, dass sich der Informationsgehalt verhältnismäßig zum Umfang des Quellcodes vergrößert. Boilerplate Code wird in wissenschaftlichen Veröffentlichungen in Zusammenhang mit Sprach- und API-Design aufgegriffen beziehungsweise fällt unter die bereits oben angesprochene aufgezwungene Duplikation oder fällt Detection-Tools als *false positive* auf, wird also im Rahmen der Klon-Forschung erwähnt, ohne als eigenständiges Phänomen untersucht zu werden.

BC wirkt auf zwei Weisen auf Lesbarkeit und Verständlichkeit von Quellcode ein. Zuträglich ist die Wiederholung von Idiomen, das sind sich wiederholende Muster, deren Funktion einer der Sprache mächtigen Person sofort offensichtlich ist (vgl. [Kapser/Godfrey 2008, S. 655]). Sind aber die BC-Fragmente sehr umfangreich und unterscheiden sich nur in kleinen, für die Bedeutung relevanten Anteilen, tritt ein negativer Effekt ein, da die zum Verständnis des Quellcodes wichtigen Zeichenfolgen vor dem Hintergrundrauschen des BC schwer auszumachen sind. Dann ist schlichtweg die zum Verständnis der Programmlogik aufzubringende kognitive Leistung höher und das Vorankommen des betreffenden Projekts verlangsamt sich. Viele Entwicklungsumgebungen bieten Möglichkeiten, häufig auftretende Sprachkonstrukte zu generieren, da die händische Erzeugung dieser Codeabschnitte selbst redundant und fehleranfällig ist (vgl. [Forward et al. 2009]). Generierung von Quellcode hilft, Flüchtigkeitsfehler bei Routineaufgaben zu vermeiden und beschleunigt so die Softwareentwicklung. Der Umfang von generiertem Code sollte dabei jedoch möglichst gering sein, da sonst andere wichtige Qualitätsfaktoren, wie die Lesbarkeit leiden können (vgl. [Lämmel/Jones 2003]). Wie bei Klonen gibt es also auch gute und schlechte BC-Fragmente, wobei die Bewertung von der Häufigkeit und der Größe des Boilerplates abhängig ist.

Im Unterschied zu DRY und Refaktorierung als Vor- und Nachsorge gegen Klone, kann eine Entwicklerin den Anteil an BC im Quellcode nur äußerst indirekt beeinflus-

sen. Das gilt sowohl für den Einsatz von vollwertigen Programmiersprachen als auch für APIs. Die Entwicklerteams der Programmiersprachen tragen hierfür die Verantwortung, da der Umfang von BC in der Syntax der Programmiersprache gründet. Wie fast jeder Quellcode unterliegen auch Programmiersprachen einer inkrementellen Weiterentwicklung, die sich in ihrer Versionierung manifestiert. Softwareentwickler können als Nutzer einer Sprache ihren Quellcode auf neuere Versionen der Sprache migrieren, die mit der Sprachgenese einhergehenden Verbesserungen umsetzen und damit ihren Code aktuell halten. Die Migration auf neuere Versionen ist ein Sonderfall von Refaktorierung, da auch hier ausschließlich die Struktur des Codes verändert wird, die Funktionalität jedoch gleich bleiben soll. Zur Vorsorge gilt es, für ein geplantes Softwareprojekt eine Programmiersprache zu wählen, die für die vorliegende Problemdomäne Quellcode mit geringem BC-Anteil zu liefern verspricht.

Obgleich die Praxisrelevanz und der Einfluss auf Codequalität des Themas Boilerplate Code besteht und zudem für Gedeih und Verderb der Popularität und Praxistauglichkeit von Programmiersprachen zentral ist, blieben eine Literaturrecherche sowie die Suche nach einer Begriffsdefinition in gängigen fachspezifischen Datenbanken weitgehend erfolglos. Wird der Begriff in Zusammenhang mit *domain specific language*- und API-Design gelegentlich verwendet, liegt ihm doch keine fundierte Definition zugrunde. Interessant ist jedoch der Kontext des Auftretens. So wird für die Erstellung von APIs geraten, durch die Wahl geeigneter Default-Werte, die Notwendigkeit des Einsatzes von all zu viel BC gering zu halten (vgl. [Blanchette 2008; Toegl et al. 2012]). Die Redundanz des BC, die an dieser Stelle entstehen würde, wird in die Ebene der API verbannt und somit vor dem Anwender der Sprache verborgen. Dieselbe Argumentation greift auch für das Design von Programmiersprachen. Dadurch wird der Overhead für die Implementierung von oft wiederholten Standardaufgaben reduziert und so an vielen Stellen längliche Codekonstrukte vermieden (vgl. [Toegl et al. 2012, S. 953]). Es obliegt den Entwicklern von Programmiersprachen, diese so zu gestalten, dass sie ihre Anwender möglichst selten zu umfangreichen Wiederholungen zwingen. Weitere Forschung bezieht sich auf Boilerplate- oder Fülltext in Dokumentationen von APIs als Gegenpol zu bedeutungsvoller Dokumentation, wo der Informationsgehalt durch Wiederholung von offensichtlichen Gegebenheiten herabgesetzt wird und damit Bedeutung einbüßt (vgl. [Watson et al. 2013]). Für die Quelltextsegmente, auf die diese Kritik trifft, liegt Redundanz zwischen Quellcode und Dokumentation vor. Von der oben erläuterten Typisierung für CCs würden diese Dopplungen übrigens gar nicht erfasst, da bereits für Typ-1-Klone Kommentierungen nicht beachtet werden.

Aktiv ignoriert wird BC von den Werkzeugen zur Clone-Detection, da BC während der Entwicklung als aufgezwungene Duplikation nicht beeinflusst werden kann. Da-

mit ist er für Refaktorierung innerhalb einer Sprachversion nicht relevant. Also sollen diejenigen Dopplungen, die aus der Programmiersprache resultieren, nicht als solche gekennzeichnet werden und treten allenfalls als *false positives* negativ auf. Es werden dann ähnliche Codesegmente von den Erkennungsalgorithmen als Klone identifiziert, deren Markierung als Klone in dem vorliegenden Anwendungsfall nicht erwünscht ist. In den Schritten, die ein Programm zur Erkennung von Klonen durchläuft, gibt es Phasen der Vor- und Nachbearbeitung, wo Codeabschnitte vorsortiert und gefiltert werden. Vor der eigentlichen Klon-Erkennung werden Codeabschnitte entfernt, die für die Klon-Findung uninteressant oder hinderlich sind. Dazu zählen auch solche Abschnitte, die mit hoher Wahrscheinlichkeit *false positives* produzieren, also auch Boilerplate Code. Automatisierte Heuristiken oder manuelle Analyse sollen zu Unrecht als Klone markierte Codefragmente erkennen und aussortieren (vgl. [Roy et al. 2009, S. 472ff]). Formal handelt es sich um Klone, praktisch ist deren Kennzeichnung als solche jedoch unerwünscht, da sie nicht eliminiert werden können. Das betrifft beispielsweise *Getter* und *Setter* in Java. Zwar handelt es sich um syntaktische Duplikate im Quellcode, jedoch können diese von der Entwicklerin nicht wegrefaktoriert werden, da die Sprache und deren gängige Konvention diese Konstrukte genau in dieser Form verlangen. Die Ähnlichkeit ist bewusst erzeugt, um der Konvention zu folgen, die gleiche Intention, nämlich einen Wert abzufragen oder zu verändern, auch auf die gleiche Weise auszudrücken. Hier werden mit Wiedererkennung durch Wiederholung sprachlicher Strukturen Verständlichkeit und Lesbarkeit erhöht, solange die betreffende Textstelle nicht zu umfangreich wird und so die Bedeutung wieder verschleiert.

Damit zeigt BC starke Parallelen zu bestimmten Fällen von als gut bewerteten Klonen oder wird in einigen Studien als Unterklasse oder Sonderfall von CCs behandelt. In einer Untersuchung von C. J. Kapser und M. W. Godfrey wird die pauschale Verteufelung von Klonen angeprangert und näher betrachtet, welche Arten von CCs, die als Clone-Patterns klassifiziert werden als gut oder eben schlecht gelten können. Ganz ähnlich der aufgezwungenen Duplikation findet sich dort ein Konzept, das BC Rechnung trägt. *„Boiler-plating Due to Language Inexpressiveness“* [Kapser/Godfrey 2008, S. 654] wird der Gruppe Templating zugeordnet und in den angeführten Beispielen zumeist als gut bewertet, da auf Anwender-Ebene der Sprache keine Alternativlösung zur Aufnahme von Redundanzen in den Quellcode zur Verfügung steht. Zu der positiven Bewertung trägt zudem der Wiedererkennungseffekt von sprachlichen oder algorithmischen Idiomen bei, die alle der Gruppe Templating zugeordneten Klone genießen. Idiome sind selbstdokumentierend, da sie allgemein bekannte, standardisierte und strukturierte Musterlösungen für Problemstellungen gleichen Typs liefern und damit die Verständlichkeit des Quellcodes verbessern (vgl. [Kapser/Godfrey 2008, S. 655f]).

Das Clone-Pattern *Verbatim Snippets* [Kapser/Godfrey 2008, S. 662] bezieht sich auf Boilerplate Code, dessen Ursache einerseits in der Beschaffenheit der Programmiersprache, zum Anderen in der Problemdomäne liegt. Dabei wiederholen sich kleine Codeabschnitte sehr häufig. Exemplarisch werden `for-Schleifen`, Fehlerbehandlung und *null-checks* angeführt, die unter die Gruppe der *exact matches* fallen, also Typ-1 oder -2-Klone sind. Java hat zur Vermeidung der redundanten *null-checks*, die genau dieses Clone-Pattern umsetzen, die Klasse `Optional<T>` eingeführt die in Kapitel 3.3 genauer beschrieben wird. Solche identischen Codestellen, die wenig eigene Semantik transportieren, werden oft kopiert. Vorteilhaft für die Verständlichkeit sind konzeptuelle Kohäsion, simpler Code und Lokalität. Alternativ können sie als wiederverwendbare Funktionalität implementiert werden, wobei sich die eben genannten Vorteile negieren und durch viele kleine Funktionalitäten aufgeblähte Interfaces resultieren können. Da kleine Klone schwer zu finden sind, ist die Änderungssynchronisation erschwert. Häufen sich *Verbatim Snippets* stark, kann die Funktionalität in eine Hilfsmethode ausgelagert werden (vgl. [Kapser/Godfrey 2008, S. 662]). Da die Sprache eine redundanzfreie Problemlösung nicht bereitstellt oder Abstraktion die Komplexität des Quellcodes zu stark erhöht, sind *Boiler-plating Due to Language Inexpressiveness* und *Verbatim Snippets* nach Abwägung der Alternativen akzeptabel (vgl. [Kapser/Godfrey 2008, S. 677]). Das Boilerplating durch Entwickler wird als gute Lösung für die technologische Limitation gesehen, da hier der Wiedererkennungswert ähnlicher Lösungskonzepte für ähnliche Problemräume die Verständlichkeit erhöht (vgl. [Kapser/Godfrey 2008, S. 654]). Nichtsdestotrotz leidet die Wartbarkeit des Codes, da die Codemenge zunimmt und die Verbreitung von Änderungen innerhalb einer Klon-Gruppe oft mit hohem Aufwand einhergeht. Der Änderungsaufwand erhöht sich um den Faktor der Anzahl der Klone in der Gruppe. Gewissenhafte Dokumentation oder Tracking der Duplikate muss sicher stellen, dass im Fall von Änderungen auch alle betreffenden Stellen bearbeitet werden. Tracking, Linked Editing oder Code Generierung können den erhöhten Änderungsaufwand wieder senken. Wird die Weiterentwicklung nicht synchronisiert, drohen schwer auffindbare Fehler in der Software (vgl. [Kapser/Godfrey 2008, S. 654]). Aus Entwicklerperspektive wird ein Workaround für in der eingesetzten Programmiersprache nicht vorhandene Sprachmittel und dessen Management angeraten. Für die betreffende Sprache kann diese Situation entweder bedeuten, dass sie auf eine Problemdomäne angewandt wird, für die die Sprache nicht optimal designed ist oder, tritt die Situation gehäuft auf, kann Evolutionsdruck für die Sprache entstehen. In neuen Versionen einer Sprache können Features angepasst oder neu eingeführt werden. Sie muss also die benötigten Sprachfeatures einführen, um fit für ihre stets dynamische Umwelt zu bleiben.

Analog zur Eliminierung von Code Clones durch Refaktorierung, kann bei der Migration auf eine höhere Version einer Programmiersprache Redundanz im Quellcode reduziert werden, weil Boilerplate Code wegfallen kann. Code Clones und Boilerplate Code haben, obwohl sie auf ganz unterschiedliche Weisen entstehen und reduziert werden können, viele Gemeinsamkeiten. Gleich ist CCs und BC die Redundanz, die sie in den Quellcode einbringen. Sie unterscheiden sich durch ihre Entstehungsgründe. Klone werden bei dem Gebrauch der Sprache durch den Entwickler in den Quellcode eingebracht, Boilerplate Code wird von der Sprache selbst verlangt. Die Anzeige von BC in Clone-Detection-Tools ist für einen Entwickler wenig zielführend, andererseits kann eine BC-Detection-Software für Sprachdesigner ein nützliches Analysewerkzeug sein. Beide Phänomene sind so ähnlich, dass hier der Versuch gewagt werden soll, ihre Gemeinsamkeiten aufzufinden und auf einer höheren Abstraktionsebene zu bündeln. Das soll im folgenden Abschnitt anhand des Begriffs von Quellcoderedundanz geschehen.

2.3 Quellcoderedundanz

Bisher wurde ein Überblick über bestehende Forschungsansätze zu Code Clones und Boilerplate Code gegeben. Dabei hat sich gezeigt, dass die Forschung stark von der Toolentwicklung getrieben ist. Eine Folge der Abwesenheit einer allgemein zugrunde gelegten Definition der Begriffe in der Forschung ist ein zwischen unterschiedlichen Forschern inkonsistenter Sprachgebrauch. Daher soll, ausgehend von alltagssprachlichen Begriffen, ein Vorschlag ihrer Definition erarbeitet werden. CCs und BC werden definiert und von einander abgegrenzt. Dann wird der Begriff Quellcoderedundanz eingeführt, der ihre Gemeinsamkeiten beschreibt. Zur Klärung der Ausgangsfrage, ob die Reduktion solcher Quellcoderedundanz ein Antrieb für die Entwicklung von Programmiersprachen ist, folgt ein Vorschlag zur Abschätzung von Zu- oder Abnahme von Quellcoderedundanz. In Kapitel 3 wird die auf Basis der Definitionen erarbeitete Operationalisierung auf Java-Quellcode angewandt.

Nach der von Roy et al. präsentierten Definition fallen auch Wendungen der Programmiersprache in den Klonbegriff, wie beispielsweise der Kopf einer `for`-Schleife. Die Bezeichnung solcher Fragmente als CCs deckt sich weder mit deren alltagssprachlichen Auffassung, noch mit deren impliziter Phänomenologie durch Clone-Detection-Tools und ist Zeuge dafür, dass die häufig verwendete Beschreibung des als Code Clone bezeichneten Phänomens zu weit ist. Auch BC fällt dann nämlich in die Klasse der Code Clones und diese fehlende Abgrenzung ist unglücklich, da sich BC und CCs gerade im Bezug auf Umgang in Praxis und Forschung stark unterscheiden. BC als Teilmenge oder Sonderfall von CCs aufzufassen, ist ein legitimes Vorgehen, jedoch messen dann

die Erkennungstools nicht, was sie vorgeben, denn BC soll in ihren Ergebnissen nicht vorkommen. Definiert ist die Ähnlichkeit von Codefragmenten als Funktion, die sich nach den Klon-Typen richtet. Operationalisiert wird sie mit Hilfe der in Kapitel 2.1 vorgestellten Erkennungstechniken. Aus der daraus resultierenden Ergebnismenge müssen jedoch in Vor- und Nachbearbeitungsschritten einige Fragmente wieder ausgeschlossen werden. Die Differenz zwischen Ergebnis und gewünschtem Resultat zeigt, dass es Codefragmente gibt, die laut Definition ähnlich sind, aber nicht der Intention der Klonerkennung folgen, häufig, weil sie nicht refaktorierbar sind. Bei diesen Kandidaten handelt es sich unter anderem um BC. Das vermeintliche Erkennungsmerkmal für Klone, die Wiederholung ähnlicher Konstrukte, eignet sich nicht für eine Definition von dem, was an CCs und BC gleichermaßen den Code unschöner macht als er sein könnte. Die Erkenntnis neuerer Forschungen, dass auch gute Klone existieren, ist ein Hinweis auf die Wertneutralität von Wiederholung, was diese als alleiniges definitorisches Merkmal der beschriebenen Problematik anzweifelbar macht. Demnach ist auch DRY nicht in allen Situationen ratsam. Idiome stellen zwar Wiederholungen dar, enthalten aber selbst Information. Eine Dopplung erhöht nicht zwangsläufig die Redundanz im Code, wenn beispielsweise durch erhöhte Lokalität Information über die Zusammengehörigkeit oder durch die Wiedererkennung derselben Muster Information über Gleichheit transportiert wird. Auch ein längerer, dafür aber sprechender Bezeichner kann die Redundanz senken. Dass, trotz vitaler Forschung an Erkennungswerkzeugen für CCs, keine belastbare Definition auffindbar ist, mag dem Umstand geschuldet sein, dass Klone oft operational, anhand der für sie bestehenden Auffindungstechniken, identifiziert und gleichsam, lediglich implizit, beschrieben werden (vgl. [Higo et al. 2008, S. 436]). Teilweise wird BC als nicht refaktorierbar mit zu CCs gezählt oder wenn es um Clone-Detection geht implizit als *false positive* aus dem Begriff ausgeschlossen. Beide Phänomene werden trotz struktureller Ähnlichkeit in der wissenschaftlichen Literatur zu dem Thema oft nicht im Zusammenhang betrachtet oder nicht sauber abgegrenzt. Im Folgenden wird der Versuch gemacht, die bisher in der Forschung verfolgte Vorgehensweise gewissermaßen von den Füßen auf den Kopf zu stellen. Statt die Definition der Begriffe aus praktischen Vorgehensweisen abzuleiten, kann es zielführend sein, von einem Begriffssystem ausgehend mit neuem Blick Impulse für die Praxis zu erlangen. Dazu folgt nun der Vorschlag für ein Begriffssystem, das die strukturellen Ähnlichkeiten von CCs und BC im Begriff der Redundanz bündelt und zugleich eine klare Abgrenzung beider liefert.

Der Redundanzbegriff wird im Kontext der Informationstechnologie oft als bewusst erzeugte Dopplung zur Absicherung gegen Systemausfälle definiert (vgl. [Hoffmann 2013, S. 98ff]). Der hier besprochene Begriff der Quellcoderedundanz ist streng von dem der Sicherheitsredundanz abzugrenzen, da Intention, Umgang und Wertung sich häufig

diametral entgegenstehen. Die nicht-fachliche Definition von Redundanz als *„Vorhandensein von eigentlich überflüssigen, für die Information nicht notwendigen Elementen“* [Dudenredaktion 2017c] umspannt die zwei besprochenen Phänomene ohne Probleme. Sie trifft den Kern der Kritik an CCs und BC recht gut, muss aber zur Beschreibung der hier vorliegenden Gegenstände noch weiter spezifiziert werden. Im Bereich der Kybernetik ist dabei Information der *„Gehalt einer Nachricht, die aus Zeichen eines Codes zusammengesetzt ist“* [Dudenredaktion 2017b]. Eine Nachricht hat einen Sender, das ist die Person, die den Code schreibt und einen Empfänger, also die Person, die den Code liest. Ihr Gehalt, also ihr *„gedanklicher Inhalt“* [Dudenredaktion 2017a] wird dabei durch das Medium des Quellcodes transportiert. Die für den Transport der Information nicht notwendigen Elemente sind, bezüglich Quellcode, Elemente des sprachlichen Ausdrucks in der betreffenden Programmiersprache. Quellcoderedundanz macht die Komplexität des sprachlichen Ausdrucks aus, die reduziert werden kann, ohne dass dabei Information eingebüßt wird. Somit kann Quellcoderedundanz als Verhältnis von der Komplexität des sprachlichen Ausdrucks zum durch diesen transportierten Informationsgehalt aufgefasst werden.

$$Quellcoderedundanz = \frac{Komplexität\ des\ Ausdrucks}{transportierte\ Information}$$

Diese Definition trägt der Erkenntnis Rechnung, dass nicht alle Wiederholungen schlecht für die Codequalität sind. Idiome einer Sprache werden oft wiederholt, was sie nach üblichem Sprachgebrauch aus der Gruppe der Klonsegmente nicht ausschließt. Sieht man jedoch den durch die Wiederholung gewonnenen Wiedererkennungswert, der zweifelsohne Information transportiert, kann im Zuge von Redundanzreduktion eine knappere Schreibweise des Idioms gefordert werden, seine Abschaffung selbst steht aber außer Diskussion. Denn nicht die Wiederholung, sondern die Redundanz scheint die Codequalität zu mindern, wobei Redundanz durchaus in Form von Wiederholung auftreten kann. Durch den Einbezug des Informationsgehaltes können positive Auswirkungen von Wiederholung in die Bewertung mit einfließen. Das gnadenlose DRY oder *„once and only once“* [Beck 1999, S. 71] wird damit in seiner Strenge verworfen, dafür jedoch die positive Formulierung *„every piece of knowledge must have a single, unambiguous, authorative representation within a system“* [Hunt/Thomas 2000, S. 27] betont. Ähnliche Sachverhalte sollen auch ähnlich ausgedrückt werden.

CCs und BC sollen nun als unterschiedliche Fälle von Quellcoderedundanz aufgefasst werden. Die Unterscheidung von refaktorierbarer und von der Programmiersprache aufgezwungener Redundanz soll als Trennlinie von Code Clones und Boilerplate Code dienen. Diese Abgrenzung trägt der impliziten Definition von Code Clones Rechnung, die durch die intensive Arbeit an Erkennungswerkzeugen entstanden ist. Damit

wird jedoch eine Einordnung von BC als Teilmenge von CCs nicht mehr haltbar. Beide Phänomene stehen, klar abgegrenzt, nebeneinander. Clone-Detection-Tools sollen genau die refaktorierbaren Codeduplikate anzeigen, aufgezwungene sollten den Sprachen Evolutionsdruck bereiten. Dennoch handelt es sich in beiden Fällen um Redundanz im Quellcode. Deshalb wird hier ein Begriff von Quellcoderedundanz als Überbegriff von CCs und BC eingeführt, der die erläuterte Trennung beider Subbegriffe anhand ihrer Ursache beizubehalten erlaubt. Um diese Abgrenzung zu schärfen, werden die unterscheidenden Merkmale in die jeweiligen Definitionen aufgenommen. Ein Code Clone ist eine Quellcoderedundanz, die beim Gebrauch einer Programmiersprache durch einen Entwickler entsteht und durch Refaktorierung beseitigt werden kann, da es in der Sprache unter Beibehaltung des Informationsgehalts eine weniger komplexe Ausdrucksweise gibt. Boilerplate Code ist Quellcoderedundanz, die durch die Beschaffenheit der Sprache erzwungen ist, da es keinen weniger komplexen Ausdruck für einen bestimmten Sachverhalt in dieser Sprache gibt. Sie kann durch Sprachevolution beseitigt werden, wenn nämlich ein Ausdruck in die Sprache eingeführt wird, der dieselbe Information durch einen simpleren Ausdruck zu übermitteln im Stande ist. In diesem Fall wird ein BC-Segment durch Neuerungen der Sprache refaktorierbar und dessen Redundanz reduzierbar. Tritt dieses Segment zudem mehr als einmal auf, ist BC durch Sprachevolution zu einem CC-Segment geworden.

Bekannt und etabliert sind diverse Maße für Umfang und Komplexität von Quellcode, auf die des Weiteren zur Operationalisierung des vorgestellten Begriffs der Quellcoderedundanz zurückgegriffen wird. Eine Quantifizierung wird im Rahmen dieses Textes nicht erarbeitet werden können, jedoch soll mit Hilfe von *Source Lines of Code* (SLOC), Anzahl der Tokens und zyklomatischer Komplexität eine begründete Abschätzung erfolgen, die gegebenenfalls Anlass zu tiefer gehenden Untersuchungen bietet. Die Maße werden im Folgenden erklärt und schließlich in Kapitel 3 auf Quellcodebeispiele in Java 7 und 8 vergleichend angewandt. Techniken zur Messung von Code Clones wurden bereits in Kapitel 2.1 vorgestellt. Wie erläutert, umfassen diese das hier auch als Quellcoderedundanz definierte Auftreten von Boilerplate Code. Zur empirischen Untersuchung von Quellcoderedundanz bieten sich zunächst etablierte Maße der statischen Code-Analyse an. Unter statischer Code-Analyse wird die Untersuchung von Quelltext verstanden, ohne dass dieser kompiliert oder ausgeführt wird (vgl. [Hoffmann 2013, S. 247]). So erhobene Kennzahlen können zuvor nicht offensichtliche Eigenschaften des Quellcodes sichtbar machen und tragen zur Qualitätssicherung bei. Des Weiteren können sie für Analyse, Vergleich und kritische Betrachtung von Programmiersprachen herangezogen werden (vgl. [Singh et al. 2011, S. 22]). Das weiter oben bereits angesprochenen grobe Maß für Programmkomplexität SLOC gehört dabei zu den leicht zu erfas-

senden, muss sich jedoch der Kritik stellen, dass es ein Gütekriterium für Maße, nämlich die Vergleichbarkeit, nur unzureichend erfüllt. Schon durch unterschiedliche Formatierung derselben Zeichenfolge, können die Werte stark schwanken, da semantische und syntaktische Aspekte in das reine Volumenmaß nicht einfließen (vgl. [Hoffmann 2013, S. 249ff]). Darum wurde für die Abschätzung des Umfangs der im Folgenden untersuchten Codesegmente auf einheitliche Formatierung geachtet. Zur Abbildung der inneren Struktur von Quellcode werden dessen Tokens gezählt, deren Aussagekraft über das Erscheinungsbild des Quelltextes hinaus geht. Dabei wird das Programm als aus Tokens bestehend aufgefasst, die in gewisser Hinsicht die Worte einer Sprache repräsentieren. Die Gesamtanzahl der Tokens setzt sich aus der Summe der absoluten Anzahl der Operatoren und der Operanden zusammen und entspricht der Programmlänge nach Halstead (vgl. [Singh et al. 2011, S. 23f]). Für Java lassen sich sechs Arten von Tokens differenzieren, wobei Kommentare eine dieser Arten sind, die hier nicht betrachtet werden sollen. Die verbleibenden fünf Source-Tokens sind Bezeichner, Schlüsselwörter, Trennzeichen, Operatoren und Literale (vgl. [Ullenboom 2006, S. 107]). Die Anzahl der Tokens, die benötigt wird, um einen bestimmten Sachverhalt auszudrücken, macht eine Abschätzung der Komplexität eines Ausdrucks möglich. Zyklomatische Komplexität hingegen ist in der Lage, semantische Merkmale des Kontrollflusses zu erfassen. Sie bildet die Anzahl von Entscheidungen ab, die während des Programmablaufs getroffen werden können. Es gibt unterschiedliche Vorgehen, die zyklomatische Komplexität zu erfassen. Ein populärer Vorschlag stammt von McCabe (vgl. [McCabe 1976]). Dabei ist die $zyklomatische\ Komplexität = |E| - |V| + 2p$, mit E als Anzahl der Kanten, V als Anzahl der Knoten und p als die Anzahl der mit dem betrachteten Codeabschnitt verbundenen Komponenten. Für ein Programm mit nur einem Einstiegspunkt gleicht sie Eins plus der Anzahl der Kontrollstrukturen oder Entscheidungspunkte, da diese Abzweigungen des Kontrollflussgraphen sind. Die zyklomatische Komplexität beschreibt die Struktur des Kontrollflussgraphen und ist somit unabhängig von der Länge des Quelltextes. Der Wert dieses Maßes korrespondiert jedoch nicht immer mit der vom menschlichen Leser empfundenen Verständlichkeit des Quelltextes, da beispielsweise `switch`-Anweisungen, formal korrekt als Verzweigung der Programmlogik gemessen werden, im Allgemeinen als gut lesbar und übersichtlich gelten (vgl. [Hoffmann 2013, S. 259], [Liggesmeyer 2002, S. 256]). Da die McCabe-Metrik nicht für funktionale Elemente wie Lambda-Ausdrücke ausgelegt ist, wird unter Abwesenheit von Kontrollstrukturen wie `if`, `switch` oder `for` immer ein Wert von eins das Ergebnis sein. Damit ist sie für die vorliegende Untersuchung nur bedingt aussagekräftig, was an den betreffenden Stellen berücksichtigt wird.

Diese Maße spiegeln nach der oben eingeführten Definition von Quellcoderedun-

danz mehr diese als allein CCs wieder. Denn Redundanzen in Quellcode sind nicht nur Klone, sondern können auch BC anzeigen. In Anlehnung an die durch das große Interesse an der sogenannten Klonerkennung bereits etablierten Messtechniken, sollen die hier vorgestellten Kennzahlen später auch auf BC angewandt werden, um die beispielhaften Codesegmente vor und nach einer Refaktorierung grob vergleichen zu können. Die Quantifizierung der übermittelten Information, die als vom Autor an den Rezipienten übertragene Bedeutung aufgefasst werden kann, gestaltet sich ungleich schwieriger. Eine exakte Quantifizierung des gedanklichen Inhalts ist nicht möglich, zumal die Aufgabe, den Bedeutungsbegriff[3] zu klären, in der Hand ganz anderer Disziplinen liegt. Zwar ist der Begriff der Quellcoderedundanz aus Überlegungen um BC und CCs entstanden, es ist aber keineswegs ausgeschlossen weitere Phänomene unter ihn zu ordnen. Bei der im Folgenden vorgestellten Messung des als Quellcoderedundanz definierten Verhältnisses ist zu beachten, dass die verwendeten Komplexitätsmaße nur grobe Annäherungen darstellen und es soll betont sein, dass der Informationsgehalt noch weniger quantifizierbar ist. Darum kann in diesem Kontext lediglich von Informationszuwachs oder -abnahme gesprochen werden, wobei selbst diese Abschätzung in vielen Fällen streitbar bleiben wird. Es wird bei nachstehender Untersuchung immer nur ein grober Trend abgeschätzt, der bestenfalls als Hinweis auf die Notwendigkeit weiterer Untersuchungen zu interpretieren ist.

3 Paradigmenwechsel mit Java 8

„It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.“ [Martin 2009, S. 48]

Mit Java 8 hat sich durch die Einführung funktionaler Sprachmittel und der Erlaubnis der Mehrfachvererbung von Programmlogik ein Paradigmenwechsel vollzogen. Für das objektorientierte Paradigma konstitutive Programmierkonzepte sind Vererbung und Polymorphie. In Java ist für Klassen nur Einfachvererbung vorgesehen, das heißt, eine Klasse darf von maximal einer Superklasse erben, aber beliebig viele Interfaces implementieren. Da diese nun auch Programmlogik enthalten können, hält mit Java 8 Mehrfachvererbung von Verhalten Einzug in die Sprache. Konzepte des funktionalen Paradigmas, wie die Parametrisierung von Verhalten und die Funktionskomposition sind seit Java 8 umsetzbar. Ob diese Zäsur in der Entwicklung Javas die Reduktion

[3] *„Wenn man in der gewöhnlichen Weise Worte gebraucht, so ist das, wovon man sprechen will, deren Bedeutung.“* [Frege 1892, S. 28]

von Redundanz im Quellcode ermöglicht oder sogar davon motiviert ist, soll im Folgenden näher betrachtet werden. Dazu werden erst einige Grundkonzepte von Java beschrieben, die Neuheiten der Version 8 vorgestellt und im Anschluss eine vergleichende Untersuchung der alten und neuen Sprachmittel im Hinblick auf deren redundanzreduzierende Wirkung angestellt. Dabei wird es von besonderem Interesse sein, inwieweit sich Möglichkeiten zur Eliminierung von CCs ergeben und BC reduziert werden kann.

3.1 Streams API

Bereits 1976 bemerkte Donald E. Knuth in einer vergleichenden Untersuchung über die Entwicklung von Programmiersprachen, dass den betrachteten Sprachen *high-level*-Datenstrukturen abseits von Arrays Kontrollstrukturen, die über die Möglichkeit zur Iteration mithilfe von Indizes hinaus gehen, fehlen (vgl. [Knuth/Pardo 1980, S. 94]). Mit dem *java development kit* (JDK) 1.8 wurde das Collections API um Streams erweitert, die eine Verarbeitung von Collections auf höherer Abstraktionsebene als zuvor ermöglichen. Dabei beschreibt der Programmcode *Was*, also welche Operationen durchgeführt werden sollen. Die technische Ausführung der Operationen, das *Wie*, ist in dem API gekapselt. `Stream<T>` ist eine Sequenz von Objekten des Typs `T`, die durch eine *Pipe*, also die Aneinanderreihung von Methodenaufrufen des Streams API verändert werden können. Bei dieser Vorgehensweise erhöht sich der Abstraktionsgrad im Vergleich zur klassischen Iteration mit Schleifen, da ganze Streams verändert werden können statt einzelner Objekte (vgl. [Urma et al. 2014, S. 8]). In der Geschichte von Java ist dies nicht die erste Weiterentwicklung der Ausdrucksmöglichkeiten für Iterationen. Listing 1 veranschaulicht die schrittweise Steigerung der Abstraktion von Iterationen in Java. Aus einer Liste von Künstlern sollen diejenigen gezählt werden, die aus London kommen. Dazu werden zum Vergleich Sprachmittel aus den Java-Versionen 1, 5 und 8 genutzt.

```
1  // a) Java 1 for-Schleife
2  int count = 0;
3  for (int i=0; i < allArtists.length; i++) {
4      if(allArtists[i].isFrom("London")) {
5          count++; } }
6  // b) Java 5 verbesserte for-Schleife
7  int count = 0;
8  for(Artist artist: allArtists) {
9      if(artist.isFrom("London")) {
10         count++; } }
11 // c) Java 8 Streams
12 long count = allArtists.stream()
```

```
13      .filter(artist -> artist.isFrom("London"))
14      .count();
```

Listing 1: Evolution der `for`-Schleife (a) in Anlehnung an [Oracle 2017], b) und c) aus [Warburton 2014, S. 21, 23])

Die klassische `for`-Schleife in Beispiel a) benötigt im Schleifenkopf die Zählvariable `i`, die im Rumpf der Schleife den Zugriff auf die in der Liste enthaltenen Elemente ermöglicht. Auf diesen werden dann die gewünschten Operationen ausgeführt. Mit dem Release von Java 5 wurde 2004 die sogenannte verbesserte `for`-Schleife in die Sprache aufgenommen die in Beispiel b) genutzt wird. Der Schleifenkopf kommt mit elf Tokens weniger aus als ihr Vorgänger und der Zugriff auf die Objekte der Liste erfolgt ohne die explizite Angabe einer Zählvariable. Das Grundgerüst einer `for`-Schleife hat in seiner ursprünglichen Form mindestens 14 Tokens allein um die Iteration zu ermöglichen. Die verbesserte `for`-Schleife kommt dafür mit sieben Tokens aus.

Mit dem Streams API können `for`-Schleifen ersetzt werden, indem die externe Iteration über Einzelobjekte in die Bibliothek internalisiert wird. Der Wegfall dieser *verbatim Snippets* macht den Quellcode kürzer, besser lesbar und sicherer (vgl. [Java Platform 2017d]). Beispiel c) implementiert dieselbe Logik mit Hilfe des in Java 8 eingeführten Streams API, die ganz von den Einzelobjekten der Collection abstrahiert. Der Aufruf von `stream()` auf die Liste der Künstler ersetzt das sperrigere `for`-Konstrukt, `filter()` implementiert die zuvor mit der `if`-Anweisung formulierte Auswahl der Londoner Künstler und `count()` übernimmt das Zählen. Das Argument der Filter-Methode ist ein Beispiel für Lambda-Ausdrücke, die in Kapitel 3.2 noch besprochen werden. Durch den Wegfall der `for-if`-Konstruktion sinkt die Schachtelungstiefe und mit ihr die zyklomatische Komplexität um zwei. Die Gliederung des Quellcodes als sequenzielle Aneinanderreihung der Operationen anstelle ihrer Verschachtelung macht den Code zudem übersichtlicher. Die Reduktion von BC, der zur Durchführung von Operationen auf allen Elementen einer Collection benötigt wird, mindert die Obfuskation der Programmlogik (vgl. [Warburton 2014, S. 21ff]). Tabelle 1 zeigt den Rückgang der Komplexität des Ausdrucks anhand der in Kapitel 2.3 vorgestellten Maße.

Beispiel	SLOC	Anzahl Tokens	zyklom. Komp.
a) Java 1	4	34	3
b) Java 5	4	23	3
c) Java 8	3	21	1
Δ	-1	-13	-2

Tabelle 1: Evolution der `for`-Schleife

Da im Zuge der Evolution der Syntax für Iterationen in Java bei konstantem Informationsgehalt ein Rückgang der Komplexität des Ausdrucks beobachtbar ist, kann nach den in Kapitel 2.3 angestellten Überlegungen eine Abnahme der Quellcoderedundanz festgestellt werden. Durch den Einsatz des Streams API werden zur Iteration weniger Tokens benötigt und so wird der Anteil an Boilerplate Code gesenkt. Die sequentielle statt einer verschachtelten Gliederung des Quellcodes macht diesen besser lesbar. Ein höheres Abstraktionsniveau wird durch interne statt externer Iteration umgesetzt. Die evolutionäre Entwicklung von Java zeigt sich in einer Reduktion von BC und einer Steigerung der Abstraktion, womit die Quellcoderedundanz gemindert wird.

3.2 Lambda-Ausdrücke

Die Parametrisierung von Verhalten ist ein Programmierkonzept, das mit Lambdas Einzug in Java 8 hielt. Funktionalität als Methodenargument weiterzugeben, ist ein Kernprinzip der funktionalen Programmierung, durch die CCs ohne die Verwendung weiterer Abstraktionsebenen und unter Wahrung der Lokalität umgangen werden können. Java wird damit um ein mächtiges, neues Idiom erweitert (vgl. [Urma et al. 2014, S. 5]). Ein Lambda ist ähnlich einer Methode ein Container für Quellcode, der ausschließlich Funktionalität beinhaltet und weder einen Namen noch einen explizit angegebenen Rückgabetyp besitzt. Die Ausdrucksmächtigkeit der Sprache wird jedoch nicht vergrößert, da alle Sachverhalte, die durch Lambdas ausgedrückt werden können, bereits zuvor in Java abgebildet werden konnten. Der Gewinn, der durch die Neuerung entsteht, bezieht sich ausschließlich auf die Art und Weise des Ausdrucks und macht unelegante Sprachkonstrukte wie anonyme innere Klassen, Methodenklone, stark verschachtelte `if`-Konstrukte und die in Listing 1 bereits vorgestellten Iterationen per `for`-Schleife an vielen Stellen obsolet. Nach den in Kapitel 2.3 erarbeiteten Definitionen werden sperrige BC-Fragmente mit JDK 1.8 refaktorierbar und der Code kann verschlankt werden. Einer Methode kann als Argument direkt eine andere Methode übergeben werden. Das spart BC ein und die Möglichkeiten zur Wiederverwendung werden offensichtlicher (vgl. [Inden 2015, S. 4]).

Listing 2 zeigt unter a) zwei Methodenklone dritten Typs, die aus einer Liste von Äpfeln einmal die großen und einmal die roten auswählen. Die Methoden unterscheiden sich nur in ihrem Namen und in ihren Filterkriterien für die Äpfel, die in den Zeilen 5 und 11 als Argumente der `if`-Statements implementiert sind. In Java werden häufig Methoden geschrieben, die sich nur in einem Statement unterscheiden, was sie zu Typ-3-Klonen macht. Diese können durch Lambdas ersetzt werden. Beispiel b) zeigt dieselbe Programmlogik in einer Zeile durch ein Lambda ausgedrückt, das beide Fil-

terkriterien in einem Ausdruck anwendet.

```
1  // a) Methodenklone
2  public static List<Apple> filterRedApples(List<Apple> inventory){
3      List<Apple> result = new ArrayList<>();
4      for (Apple apple: inventory){
5          if ("red".equals(apple.getColor())) {
6              result.add(apple); } }
7      return result; }
8  public static List<Apple> filterBigApples(List<Apple> inventory){
9      List<Apple> result = new ArrayList<>();
10     for (Apple apple: inventory){
11         if (apple.getWeight() > 150) {
12             result.add(apple); } }
13     return result; }
14 // b) Lambda-Ausdruck
15 filterApples(inventory, (Apple a) -> a.getWeight() > 150 &&
16     "red".equals(a.getColor()));
```

Listing 2: Parametrisierung von Verhalten mit Lambdas (in Anlehnung an [Urma et al. 2014, S. 14f])

Von den Filterkriterien abgesehen wird der Code für die Iteration über die Liste und die Durchführung der Filterung selbst benötigt. Dieser Boilerplate-Anteil kann durch eine Refaktorierung von Methodenklonen in einen Lambda-Ausdruck deutlich reduziert werden, wie in Tabelle 2 zusammengefasst ist. Für die Implementierung der Methoden werden 96 Tokens benötigt. Wird dafür ein Lambda genutzt, reichen 24 Tokens aus.

Beispiel	SLOC	Anzahl Tokens	zyklom. Komp.
a) Methodenklone	12	96	6
b) Lambda	1	24	1
Δ	-11	-72	-5

Tabelle 2: Parametrisierung von Verhalten mit Lambdas

Da das Lambda gleich mehrere Kriterien kombiniert auswerten kann, bleibt der ohnehin schon geringe BC-Anteil auch bei Hinzunahme weiterer Kriterien konstant. Für die Implementierung mit Methoden kommt für jedes Filterkriterium ein Klon hinzu. Der variable Skalierungsfaktor n für die Komplexitätsänderung ist hier die Anzahl der Filterkriterien. Die zyklomatische Komplexität sinkt bei der in Listing 2 gezeigten Migration auf Java 8 um $3n$. Der Boilerplate Code reduziert sich um 72 Tokens. Die

anhand von Listing 1 gezeigte Problematik der schlechten Lesbarkeit von Verschachtelungen betrifft auch den Code aus Listing 2. Für das Verständnis zentrale Codesegmente sind die Filterkriterien, die auf unterster Schachtelungsebene der `for-if`-Konstruktion stehen. Gerade bei ähnlichen Methoden kann der sie unterscheidende Code in Form von Lambdas als Argument mitgegeben werden, sodass die Versuchung des *Copy, Paste and Modify*, das einen Klon erzeugen würde, umgangen wird. Der Quellcode wird durch die Parametrisierung von Verhalten wesentlich kürzer, klarer und weniger fehleranfällig. Die Ausdruckskomplexität kann erstens durch die Sublimierung einer Methode mit `for-if`-Konstruktion durch einen einzeiligen Lambda-Ausdruck gesenkt werden. Zweitens durch den Wegfall von Typ-3-Methodenklonen, weil ein Lambda-Ausdruck beliebig viele Filterkriterien in Kombination behandeln kann.

Eine weitere Refaktorierung, die durch Java 8 ermöglicht wird, ist der Einsatz von Lambdas anstelle von anonymen inneren Klassen (vgl. [Warburton 2014, S. 9]). Java erlaubt Klassen innerhalb von Klassen. Werden sie nur dort benötigt, erhöht dies die Kapselung und die Lokalität. Vor der Möglichkeit Lambdas einzusetzen, wurde Verhaltensparametrisierung oft mit anonymen inneren Klassen umgesetzt, die aber viel BC für relativ wenig Informationsgehalt bedeuten. Das Listing 3 zeigt, dass dieselbe Funktionalität durch Lambdas sehr viel kürzer und klarer ausgedrückt werden kann (vgl. [Urma et al. 2014, S. 5]). Verhaltensparametrisierung fördert die Wiederverwendung von bereits geschriebenem Code, da Variationen die, ansonsten eine ähnliche Neuimplementierung verlangten, nun als Argument an bereits bestehende Methoden weitergeben werden können (vgl. [Urma et al. 2014, S. 9]). Anonyme innere Klassen sind Ausdrücke, die innerhalb eines anderen Ausdrucks ohne Namen gleichzeitig deklariert und instanziiert werden. Dieses Vorgehen ist üblich, wenn das durch die anonyme innere Klasse ausgedrückte Verhalten nur einmalig gebraucht wird. Sie selbst und ihre Member dürfen nicht als `static` deklariert werden und sind somit frei von Seiteneffekten. Diese Konstrukte können gut durch Lambdas ersetzt werden, was sie unter Beibehaltung der Funktionalität syntaktisch leichtgewichtiger macht.

```
1 // a) anonyme innere Klasse
2 button.addActionListener(new ActionListener()) {
3     public void actionPerformed(ActionEvent event) {
4         System.out.println("button clicked"); } };
5 // b) Lambda
6 button.addActionListener(event -> System.out.println("button
     clicked"));
```

Listing 3: Lambda statt anonymer innerer Klasse [Warburton 2014, S. 12]

Wie Listing 3 illustriert, bietet diese Refaktorierung eine Reihe von Vorteilen und neuen Ausdrucksmöglichkeiten. Der BC-Anteil verringert sich und zudem wird die Intention durch die Weitergabe von Verhalten klarer. Statt ein das Interface implementierendes Objekt in die Methode zu geben, wird dort mit der anonymen Methode ein Stück Code hineingereicht. Auch der Typ von `event`, nämlich `ActionListener`, muss hier nicht angegeben werden, da er sich für den Compiler aus der Signatur von `addActionListener` ergibt. Soll der Typ jedoch explizit gemacht werden darf er dazugeschrieben werden. Gleiches gilt für den Umstand, dass Variablen in Lambdas zwar effektiv `final` sein müssen, also nur einmal einen Wert zugewiesen bekommen dürfen, der sodann unveränderlich ist. Das Schlüsselwort `final` ist optional. Es werden schließlich ganz im funktionalen Stil Werte und keine Variablen in Lambda-Ausdrücken genutzt. An solchen Stellen gewährt Java 8 seinen Nutzern mehr Freiheit als die Vorgängerversionen. Die Nutzer der Sprache sind frei, solche Sachverhalte, die sich aus dem Kontext ablesen lassen, zwecks besserer Verständlichkeit explizit anzugeben oder sie zugunsten der Kürze des Ausdrucks nicht zu notieren (vgl. [Warburton 2014, S. 14]). Tabelle 3 zeigt die Reduktion von BC durch den Einsatz eines Lambda-Ausdrucks anstelle einer anonymen inneren Klasse.

Beispiel	SLOC	Anzahl Tokens	zyklom. Komp.
a) anonyme innere Klasse	3	24	1
b) Lambda	1	14	1
Δ	-2	-10	0

Tabelle 3: Lambda statt anonymer innerer Klasse

Der in Listing 3 benötigte BC sinkt durch die Refakorierung von Beispiel a) zu b) um zehn Tokens. Der Skalierungsfaktor ist bei der Umformung einer anonymen inneren Klasse in einen Lambda-Ausdruck immer eins, da eine Voraussetzung für diese Konstrukte eben die einmalige Nutzung der Funktionalität ist. Hier wird also nicht wie in Listing 2 ein Konstrukt für diverse Klone eingetauscht, sondern ein BC-lastiges Konstrukt durch ein leichtgewichtigeres ersetzt. Wird die Anzahl der anonymen inneren Klassen als Skalierungsfaktor n angenommen, ergibt sich eine Reduktion um $10n$ Tokens auf $2n$ SLOC. Die Abschätzung gilt für das hier gezeigte Beispiel und kann mit dem Umfang der Klasse variieren. Da ein Minimalbeispiel vorliegt, kann allgemein von einer Reduktion der Komplexität des Ausdrucks ausgegangen werden.

Für die in Listing 2 und 3 vorgestellten Refaktiorierungen häufig genutzter Sprachkonstrukte in Java zu Lambda-Ausdrücken konnte beispielhaft eine Reduktion der

Quellcoderedundanz gezeigt werden. Listing 2 veranschaulicht den Wegfall von Methodenklonen und dessen Skalierung. Je größer die Klongruppe ist, desto mehr Ausdruckskomplexität kann durch den Einsatz von Lambdas eingespart werden. Zusätzlich gehen mit den CCs die mit ihnen verbundenen Risiken und Probleme, die in Kapitel 2.1 diskutiert wurden, verloren. Listing 3 zeigt die Möglichkeit des Abbaus von BC durch den Wegfall innerer anonymer Klassen. Lambda-Ausdrücke sind nicht nur ein für Java neues Idiom, sie bringen das Konzept der Verhaltensparametrisierung in die Sprache und ermöglichen die Reduktion von Quellcoderedundanz. Dass dieser Effekt sogar die Motivation hinter der Einführung des neuen Konzepts sein mag, kann an den vorgestellten Beispielen nicht festgemacht werden. Die Ergebnisse können jedoch als Hinweis auf die Möglichkeit entsprechender Resultate weiterer Untersuchungen gedeutet werden.

3.3 Die Klasse `Optional<T>`

Vor Java 8 waren sogenannte *Null-Checks* omnipräsenter Boilerplate Code und zählen als *verbatim Snippets* zur Klasse der aufgezwungenen Duplikation. Dabei wird vor Ausführung einer Operation auf einer Referenzvariablen deren Existenz überprüft. Durch eine `if`-Anweisung wird explizit sichergestellt, dass die Referenzvariable auf ein Objekt verweist und nicht `null` ist. Diese Existenzprüfungen sind notwendig, da die *Java Virtual Machine* (JVM) ansonsten die berüchtigte `Null-Pointer-Exception` wirft und das Programm abbricht, wenn auf ein Objekt zugegriffen werden soll, das lediglich deklariert, aber nicht instantiiert wurde. Da Null-Checks bei jeder Verwendung des Objekts durchgeführt werden müssen, sind sie aufgrund ihrer Häufigkeit nicht nur während der Entwicklung lästig, sondern können die Lesbarkeit des Quellcodes beeinträchtigen, wie Listing 4 demonstriert. Seit Java 8 gibt es die Klasse `java.util.Optional<T>`, die mit *Haskells* `Maybe`-Typ und *Scalas* `Option[T]` von funktionalen Sprachen inspiriert ist (vgl. [Java Platform 2017f]). `Optional<T>` kapselt das Objekt in einen Container, der einen Einzelwert enthalten oder leer sein kann. `Optional<T>` kommt mit einer Reihe von Methoden, die eine dezidierte Behandlung beider Fälle erlauben (vgl. [Inden 2015, S. 219f]). In Listing 4 wird die Version der Soundkarte eines Computers unter a) mit Null-Checks und unter b) mit `Optional<T>` erfragt.

```
1 // a) Null-Checks
2 public class Computer {
3   private Soundcard soundcard;
4   public Soundcard getSoundcard() { ... } }
5 String version = "UNKNOWN";
6 if(computer != null) {
```

```
7    Soundcard soundcard = computer.getSoundcard();
8    if (soundcard != null) {
9        version = soundcard.getVersion(); } }
10 // b) Optional<T>
11 public class Computer {
12   private Optional<Soundcard> soundcard;
13   public Optional<Soundcard> getSoundcard() { ... } }
14 String version = computer.flatMap(Computer::getSoundcard)
15     .map(Soundcard::getVersion)
16     .orElse("UNKNOWN");
```

Listing 4: `Optional<T>` statt Null-Check [Java Platform 2017f]

Listing 4 zeigt die Deklaration der Membervariable `soundcard` und den zugehörigen Getter in der Klasse `Computer`. Weiter wird die Version der Soundkarte des Computers ermittelt, sofern alle dazu benötigten Objekte nicht `null` sind. Zunächst ist dies in den Zeilen 2-9 mit Java 7 als Null-Check umgesetzt und ab Zeile 11 durch `Optional<T>` aus Java 8. Durch die Verwendung von `Optional<T>` kann für den Beispielcode in Listing 4 eine Ersparnis von zwei SLOC, acht Tokens und ein Rückgang der zyklomatischen Komplexität um zwei festgestellt werden, wie Tabelle 4 zeigt.

Beispiel	SLOC	Anzahl Tokens	zyklom. Komp.
a) Null-Checks	8	46	4
b) `Optional<T>`	6	38	2
Δ	-2	-8	-2

Tabelle 4: `Optional<T>` statt Null-Check

Für die Deklaration jeder Membervariable und -methode werden für die Kapselung in ein `Optional<T>` im Vergleich zum Null-Check zwei zusätzliche Tokens benötigt. Das wird durch den Vergleich der Codeabschnitte in den Zeilen 3-4 und 12-13 deutlich. Diese Zunahme von BC wird jedoch bei deren Verwendung kompensiert, wie ein Vergleich der Zeilen 5-9 und 14-16 aus Listing 4 deutlich macht. Jedes `if`-Statement aus Beispiel a) benötigt eine SLOC, sechs Tokens und erhöht die zyklomatische Komplexität um eins. Der Skalierungsfaktor n ist die Anzahl der Verwendungen von Instanzen eines Objekts. In Beispiel a) müssen n `if`-Statements für den Null-Check ausgeführt werden, um eine `Null-Pointer-Exception` ausschließen zu können. Die zyklomatische Komplexität skaliert also mit $1n$. Dazu kommt als Konstante eine SLOC in Zeile 5 mit fünf Tokens. Für Beispiel b) gilt ein anderer Skalierungsfaktor. Eine durch die zyklomatische Komplexität nach McCabe nicht erfasste Verzweigung der Programmlogik ergibt sich

durch das `orElse()` bei der Verwendung von `Optional<T>` in Zeile 16. Dieser Zweig der Programmausführung wird eingeschlagen, wenn eine aller in der Pipe enthaltenen und durch `Optional<T>` gekapselten Instanzen `null` ist. Was geschehen soll, wenn ein Objekt `null` ist, muss nicht pro Verwendung einer Instanz, sondern nur je ein Mal pro Pipe implementiert werden.

Durch die Verwendung von `Optional<T>` kann Quellcode eingespart werden und es wird explizit angezeigt, dass das Objekt `null` sein kann. Mit einem kürzeren Ausdruck wird sogar mehr Information übermittelt. Die Quellcoderedundanz wird gemindert und die Vernebelung des Quellcodes durch Null-Checks ist nicht mehr nötig. Null-Checks sind durch die Einführung von `Optional<T>` mit dem JDK 1.8 refaktorierbar geworden und fallen laut der in Kapitel 2.3 vorgeschlagenen Definitionen nicht mehr unter BC, sondern sind zu CCs geworden.

3.4 `default`-Methoden in Interfaces

Ein angestrebtes Qualitätsmerkmal für Quellcode ist seine leichte Erweiterbarkeit. Für Interfaces in Java ergibt sich daraus ein grundsätzliches Problem. Alle Klassen, die ein Interface Implementieren, müssen für alle dort deklarierten Methoden eine Implementierung zur Verfügung stellen. Soll nun das Interface um eine neue Methode erweitert werden, kompilieren diese Klassen solange nicht, bis sie die Implementierung der Methode leisten. Handelt es sich um ein weit verbreitetes Interface, ist der Änderungsaufwand enorm. Ein häufig genutzter Workaround ist die Einführung eines zweiten Interfaces, das die zusätzlichen Funktionsdeklarationen beinhaltet und nach Bedarf von den Klassen implementiert werden kann. Für jede gewünschte Erweiterung eines Interfaces entsteht somit ein weiteres Interface (vgl. [Inden 2015, S. 12]). So kann bei lange bestehenden oder häufig erweiterten APIs ein ganzes Sammelsurium an Interfaces entstehen. Da auch hierbei für alle Klassen, die das Interface nutzen, die Implementierung seiner Methoden obligatorisch ist, entstehen Methodenklone mit den diskutierten negativen Effekten auf die Codequalität. Mit dieser Problematik sahen sich die Entwickler von Java bei der bereits in Kapitel 3.1 erwähnten Erweiterung des Collections API konfrontiert, doch konnten sich behelfen. Mit Java 8 wurde die Sprache um die Möglichkeit von bereits im Interface komplett implementierten Methoden erweitert. Diese werden nicht nur deklariert, sondern haben einen Methodenrumpf, der Programmlogik enthält. Solche Methoden werden mit dem Schlüsselwort `default` gekennzeichnet. Die Methodenimplementierung wird dadurch aus den Klassen in das Interface verlagert. Da Interfaces, anders als Klassen, mehrfach vererbbar sind, bedeutet dies den Einzug der Mehrfachvererbung von Programmlogik in Java. Beide Wege

der Erweiterung von Interfaces sind in Listing 5 zum Vergleich dargestellt. Es zeigt unter a) den Quellcode, der ohne die `default`-Methoden nötig gewesen wäre, um das Collections API um Streams zu erweitern. Das sind das zusätzliche Interface mit der Methodendeklaration und die entsprechende Implementierung der Methode in einer Klasse, die das Interface nutzt. Beispiel b) aus Listing 5 zeigt die `default`-Methode `stream()`, die für Java 8 in das Interface `Collection<E>` aufgenommen wurde. In allen Klassen, die `Collection<E>` implementieren, kann die `default`-Methode `stream()` ohne Anpassung der Klassen genutzt werden (vgl. [Java Platform 2017a]).

```
1 // a) Methodendeklaration im Interface
2 public interface Collection2<E> extends Collection<E> {
3     Stream<E> stream(); }
4 // Implementierung der Methode in einer Klasse
5 public Stream<E> stream() {
6     return StreamSupport.stream(spliterator(), false); }
7 // b) default-Methode im Interface
8 default Stream<E> stream() {
9     return StreamSupport.stream(spliterator(), false); }
```

Listing 5: `default`-Methode statt weiteres Interface (a) in Anlehnung an [Java Platform 2017a], b) aus [Java Platform 2017c])

Java 8 bietet also eine elegante Möglichkeit für die Erweiterung von Interfaces, da Änderungen, die viele Klassen betreffen, an nur einer Stelle vorgenommen werden müssen. Die in Tabelle 5 angegebenen Maße beziehen sich auf den Minimalfall, in dem nur eine Klasse das Interface implementiert. Die dort abgebildete Reduktion der Ausdruckskomplexität ist darum auf das eingesparte Interface zurückzuführen. Der Wegfall, möglicherweise sehr vieler Methodenklone, ist nicht berücksichtigt.

Beispiel	SLOC	Anzahl Tokens	zyklom. Komp.
a) Methodendeklaration	4	33	1
b) `default`-Methode	2	17	1
Δ	-2	-16	0

Tabelle 5: `default`-Methode statt weiteres Interface

Die Tragweite der Neuerung wird klar, sobald der Änderungsaufwand für viele Klassen betrachtet wird. Der Aufwand an Quellcode für `Collection2<E>` beläuft sich auf die für die einmalige Implementierung des Interfaces mitsamt der Methodendeklaration i benötigten Tokens. Zusätzlich muss für jede Klasse, die das Interface implementiert die

Methodenimplementierung geleistet werden. Wird die Anzahl der Klassen, die das Interface implementieren k genannt und die Anzahl der für eine Implementierung der Methode notwendigen Tokens m, entsteht ein Erweiterungsaufwand von $i + k * m$ $Tokens$. Die in Beispiel b) gezeigte `default`-Methode ermöglicht die Erweiterung des Interfaces mit einem konstanten Aufwand von m Tokens. Sie muss nur einmalig im zu erweiternden Interface implementiert werden und die betreffenden Klassen müssen nicht angepasst werden. Die Auswirkung auf die Funktionalität des Quellcodes ist in beiden Fällen gleich. Durch die Nutzung einer `default`-Methode anstelle eines weiteren Interfaces entsteht eine Differenz von $m - (i + k * m)$ $Tokens$. Im Fall von `Collection<E>` nutzen 34 Klassen die `default`-Implementierung von `stream()`. Für die Lösung mit einem zusätzlichen Interface `Collection2<E>` hätte das einen immensen Änderungsaufwand sowie einen starken Anstieg der Ausdruckskomplexität bedeutet. Durch die Erweiterung des Interfaces `Collection<E>` um die Methode `stream()` als `default` ergibt sich gegenüber der Implementierung eines zusätzlichen Interfaces `Collection2<E>` eine Änderung der Ausdruckskomplexität von $17 - (16 + 34 * 17)$ $Tokens = -577$ $Tokens$. Die 34 Methodenrealisierungen in den Klassen würden als Typ-1-Klongruppe in den Quellcode eingehen. Da `Collection<E>` vier `default`-Methoden hat, würden sie jeweils eine solche Klongruppe provozieren. Damit ist exemplarisch gezeigt, dass die Einführung von `default`-Methoden in Interfaces mit Java 8 deren Erweiterung im Vergleich zu den Möglichkeiten von Java 7 die Quellcoderedundanz absenken kann.

Alle durch den Implementierungszwang von Interface-Methoden entstandenen Typ-1-Klone sind seit der Einführung der `default`-Methoden obsolet. Diese Modifikation wurde nicht vorgenommen, um Java für das funktionale Paradigma zu öffnen. Sie ermöglicht die Erweiterung von Interfaces mit deutlich verringerter Quellcoderedundanz, da sie auf einer höheren Abstraktionsebene vorgenommen werden kann. Die Einführung von Methodenimplementierungen in Interfaces bedeutet für Java einen Bruch mit dem vormaligen Verbot der Mehrfachvererbung von Funktionalität. Über die Beweggründe dahinter kann nur spekuliert werden. Jedoch scheint die Bereitstellung der Änderungs- und Erweiterungsfreundlichkeit von Interfaces, respektive deren Potential für eine starke Senkung von Quellcoderedundanz, das Festhalten am Dogma der Einfachvererbung überwogen zu haben (vgl. [Java Platform 2017e; Inden 2015, S. 18]).

4 Schluss

„Niemals sollte man die logische Richtigkeit der Kürze des Ausdrucks opfern. Deshalb ist es von großer Wichtigkeit, eine mathematische Sprache zu schaffen, die mit strengster Genauigkeit möglichste Kürze verbindet.“ [Frege 1904, S. 666]

Ist die Reduktion von Quellcoderedundanz ein Motivator für die Evolution von Programmiersprachen? Diese Frage konnte aufgrund der stichprobenartigen Untersuchung nicht allgemeingültig beantwortet werden. Dazu wäre die Prüfung weiterer Sprachmittel und Sprachen über mehrere Versionen hinweg nötig. Sie konnte jedoch für Java 8 bejaht werden.

Die Anwendung der in Kapitel 2.3 entworfenen Definitionen auf die in Kapitel 3 vorgestellten Beispiele kann als erfolgreich eingeordnet werden. Zunächst, weil aus ihrer Anwendung keine Widersprüche hervorgegangen sind und weiter, weil sie in der Literatur gefundene Einschätzungen widerspiegelt. Ein Begriff von Quellcoderedundanz, der die Komplexität des Ausdrucks zum Informationsgehalt in Beziehung setzt, soll nach Möglichkeit zum weiteren Ausbau eines Theorieapparats dienlich sein. So ist die Definition weiterer Formen von Quellcoderedundanz denkbar. Dabei ist die aufgestellte Formel als ein erster Versuch der Operationalisierung zu deuten, wobei die Quantifizierung der Ausdruckskomplexität das Potential besitzt, noch präzisiert zu werden. Der Ansatz, das Ausmaß vorhandener und reduzibler Quellcoderedundanz zu quantifizieren, kann bei der konkreten Abschätzung der Nützlichkeit von Neuerungen in Programmiersprachen oder Refaktorierungen von Quellcode jenseits theoretischer Forschung nützlich gemacht werden. Die Anzeige von BC in Clone-Detection-Tools ist für einen Entwickler wenig zielführend, andererseits kann eine Boilerplate-Detection Software für Sprachdesigner ein nützliches Analysewerkzeug sein. Sobald durch Weiterentwicklung der Sprache eine Refaktorierung möglich wird, ist es angemessen, die entsprechenden Codeabschnitte in Clone-Detection-Tools nicht mehr als *false positives* aus der Ergebnismenge explizit auszuschließen, sondern als Klone anzuzeigen. So kann der theoriegetriebene Definitionsvorschlag wichtige Implikationen für verschiedene Akteure der Praxis liefern und im Idealfall neue Ansätze der Problemlösung motivieren.

Die mit Java 8 eingeführten Sprachkonstrukte können mit diversen Qualitätsverbesserungen des Quellcodes in Verbindung gebracht werden. So erhöht die Nutzung des Streams API die Lesbarkeit, die Evolution der `for`-Schleife zeigt die Abstraktion von Standardfunktionalität in das Sprach-API, wodurch deren Performance sichergestellt wird, die Nutzung von `Optional<T>` ermöglicht mit der Kontrolle über `Null-Pointer-Excepitons` verbesserte Sicherheit und die Einführung von `default`-Methoden in Interfaces vereinfacht deren Erweiterbarkeit. Damit ermöglichen die neuen Sprachfeatures ganz unterschiedliche Qualitätsverbesserungen des Quellcodes. Ihnen gemein ist die Bereitstellung von Programmierkonzepten, die es ermöglichen, bereits mit Java 7 abbildbare Funktionalität ohne Absenkung des Informationsgehalts mit reduzierter Ausdruckskomplexität zu implementieren. Wie die Untersuchung aus

Kapitel 3 gezeigt hat, sind sie alle dazu geeignet die Quellcoderedundanz zu senken. Iterationen, Prüfung auf `null`-Werte und Verhaltensparametrisierung sind durch sie mit deutlich reduziertem BC-Anteil umsetzbar. Weiter konnte gezeigt werden, dass Lambda-Ausdrücke und `default`-Methoden in Interfaces das Potential besitzen, eine Großzahl von Klonen obsolet zu machen. Dass die Vermeidung von Quellcoderedundanz bei dem mit Java 8 vollzogenen Paradigmenwechsel ein zentraler Antrieb war, lässt sich auf dieser Basis zwar nur spekulieren, aber dass sie ein notwendiges Qualitätsmerkmal für neue Sprachfeatures ist, darf zunächst für Java 8 angenommen werden. Doch die Evolution der Iterationskonzepte in Java deutet auf eine versionsübergreifende Entwicklung hin. Über die hier behandelten Beispiele hinaus lassen sich weitere Entwicklungen in Javas Sprachgeschichte finden, die das Potential zur Vermeidung von Quellcoderedundanz steigern. Die vom Compiler automatisch geleistete Typdeduktion und die Unterstützung von Methodenreferenzen sind weitere Beispiele aus Java 8, die ebenfalls in der Lage sind, Ausdruckskomplexität zu verringern. Mit Version 10 wird in Java die automatische Deduktion des Typs für lokale Variablen eingeführt. Laut *JDK enhancement proposal* 286 ist sie maßgeblich durch die mit ihr einhergehende Reduktion von BC motiviert (vgl. [Java Platform 2017b]). Es lassen sich also über Java 8 hinaus Anhaltspunkte für die Reduktion von Quellcoderedundanz als Antrieb der Evolution der Sprache finden.

Wie jeder Quellcode unterliegen auch Programmiersprachen einem Alterungsprozess und müssen, um nicht zu veralten und abgelöst zu werden, gewartet und modernisiert werden, um den ständig verändernden Anforderungen und Trends von Hardware, Entwicklern und Problemdomänen gerecht zu werden. Sie konkurrieren im Ökosystem der Sprachen um Relevanz und Verbreitung. Ein wichtiger Faktor ist dabei, dass die Sprache es ermöglicht, qualitativ hochwertigen Quellcode zu verfassen. Entstehen durch die vorgegebene Sprachsyntax große BC-Segmente oder viele CCs, kann dies als Evolutionsdruck auf die betreffende Programmiersprache interpretiert werden. Eine geringe Quellcoderedundanz zu ermöglichen könnte als mindestens notwendige Bedingung für Neuerungen in Programmiersprachen angenommen werden. Bei deren Evolution durch Versionierung wäre sie dann immer eine Anforderung. Dieser Mechanismus könnte sich als Grund für eine evolutionäre Reduktion der Quellcoderedundanz herausstellen. Java eröffnet seinen Nutzern mit der Inklusion des funktionalen Paradigmas in eine traditionell objektorientierte Sprache mehr Freiheitsgrade bei den Möglichkeiten der Modellierung von Sachverhalten. Mit *Frege* existiert bereits eine rein funktionale Sprache in der JVM, doch das *Entweder Oder* der Paradigmen scheint den von Problemstellungen der Praxis geforderten Lösungsmöglichkeiten nicht immer gerecht zu werden. Schließlich wurde mit der Erlaubnis der Mehrfachvererbung auch das Paradigma der Objektori-

entierung gestärkt, sodass die Vorteile beider Paradigmen mehr als zuvor ausgeschöpft werden können. Für das Vorankommen von Sprachen ergibt sich daraus die Zielsetzung, diese Möglichkeiten zu schaffen und für die Anwender der Sprachen ist es geboten, diese Möglichkeiten zu nutzen. So hat vielleicht der Streit der Paradigmen in der Maxime, die Quellcoderedundanz gering zu halten, einen Prüfstein gefunden. Gerade weil funktionale und objektorientierte Programmierung sehr unterschiedliche Ansätze darstellen, die verschiedene Denk- und Herangehensweisen repräsentieren, kann die Entscheidung, Java zur Multiparadigmen-Sprache auszubauen, klug genannt werden. Nutzer der Sprache sind frei, den besten Lösungsansatz aus der Welt der Objektorientierung und der funktionalen Programmierung zu wählen. Als Auswahlkriterien bieten sich dabei die Qualitätsmerkmale für Quellcode an, die gegeneinander abgewogen werden. Bei ansonsten gleichen Qualitätsmerkmalen, ist es anzuraten, dasjenige Programmierkonzept zu wählen, dessen Anwendung die Formulierung mit der geringeren Quellcoderedundanz zulässt. Weiterführend stellt sich die Frage, ob sich langfristig von zwei existierenden Programmierkonzepten dasjenige, mit dem Potenzial für eine weniger redundante Ausdrucksweise durchsetzt.

Was der Philosoph Gottlob Frege über die Mathematik schreibt, deren Sprache er in obigem Zitat betrachtet, kann auch für Programmiersprachen gefordert werden. Die Zielsetzung, dort in möglichster Kürze strengste Genauigkeit auszudrücken, ist gleichbedeutend mit der Einordnung geringer Quellcoderedundanz als Qualitätsmerkmal. Das *„Vorhandensein von eigentlich überflüssigen, für die Information nicht notwendigen Elementen“* [Dudenredaktion 2017c] zu mindern, konnte für Java 8 als ein Antrieb für Erneuerung bestätigt werden. Kann diese Erkenntnis auf Programmiersprachen im Allgemeinen erweitert werden, birgt sie neue Sichtweisen auf die Evolution von Programmiersprachen und kann damit Impulse für Forschung und Praxis liefern.

Literaturverzeichnis

Arefin, M., Khatchadourian, R., „Porting the NetBeans Java 8 enhanced for loop lambda expression refactoring to eclipse“. in: *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

Baker, B. S., „A program for identifying duplicated code“, in: *Computing Science and Statistics*, 1993, S. 49–49.

Baker, B. S., „On finding duplication and near-duplication in large software systems“. in: *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, 1995.

Balzert, H., *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Spektrum Akademischer Verlag, Heidelberg, 2011.

Beck, K., „Embracing change with extreme programming“, in: *Computer*, 1999, S. 70–77.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E., „Comparison and evaluation of clone detection tools“, in: *IEEE Transactions on software engineering*, 2007,

Blanchette, J., „The little manual of API design“, in: *Trolltech, Nokia*, 2008,

Cunningham, W. (2018), *Technical Debt*, URL: `http://wiki.c2.com/?TechnicalDebt` (besucht am 15. 01. 2018).

Duala-Ekoko, E., Robillard, M. P., „Clone region descriptors: Representing and tracking duplication in source code“, in: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2010, S. 3.

Dudenredaktion (2017a), *„Gehalt“ auf Duden online*, URL: `https://www.duden.de/node/646443/revisions/1367379/view` (besucht am 21. 03. 2017).

Dudenredaktion (2017b), *„Information“ auf Duden online*, URL: `https://www.duden.de/node/677544/revisions/1622138/view` (besucht am 21. 03. 2017).

Dudenredaktion (2017c), *„Redundanz“ auf Duden online*, URL: `https://www.duden.de/node/690182/revisions/1631573/view` (besucht am 21. 03. 2017).

Forward, A., Lethbridge, T. C., Brestovansky, D., „Improving program comprehension by enhancing program constructs: An analysis of the Umple language“. in: *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, 2009.

Fowler, M., Beck, K., *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Westford, 1999.

Frege, G., „Über Sinn und Bedeutung“, in: *Zeitschrift für Philosophie und philosophische Kritik*, 1892, S. 25–50.

Frege, G. (1904), „Was ist eine Funktion?“, in: *Festschrift Ludwig Boltzmann gewidmet zum sechzigsten Geburtstage*, Leipzig: Amrosius Barth, S. 656–666.

Gallardo, D., „Refactoring for everyone-How and why to use Eclipse's automated refactoring features“, in: *IBM developerWorks Technical library*, 2003, S. 1–20.

Higo, Y., Kusumoto, S., Inoue, K., „A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system“, in: *Journal of Software: Evolution and Process*, 2008, S. 435–461.

Higo, Y., Hotta, K., Kusumoto, S., „Enhancement of CRD-based clone tracking“. in: *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, 2013.

Hoffmann, D. W. (2013), „Statische Code-Analyse“, in: *Software-Qualität*, Springer, S. 247–332.

Hou, D., Jablonski, P., Jacob, F., „CnP: Towards an environment for the proactive management of copy-and-paste programming“. in: *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, 2009.

Hunt, A., Thomas, D., *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley Longman, Redwood City, 2000.

Inden, M., *Java 8 Die Neuerungen, Lambdas, Streams, Date and Time API und JavaFX im Überblick.*, Heidelberg, 2015.

Jablonski, P., Hou, D., „CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE“. in: *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007.

Java Platform (2017a), *Evolving Interfaces*, URL: `https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html` (besucht am 13. 02. 2017).

Java Platform (2017b), *Evolving Interfaces*, URL: `http://openjdk.java.net/jeps/286` (besucht am 21. 02. 2017).

Java Platform (2017c), *Java SE Development Kit 8 - Downloads*, URL: `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html` (besucht am 13. 02. 2017).

Java Platform (2017d), *JEP 126: Lambda Expressions & Virtual Extension Methods*, URL: `http://openjdk.java.net/jeps/126` (besucht am 12. 04. 2017).

Java Platform (2017e), *Multiple Inheritance of State, Implementation, and Type*, URL: `https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html` (besucht am 11. 09. 2017).

Java Platform (2017f), *Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional!*, URL: `http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html` (besucht am 13. 02. 2017).

Jiang, L., Misherghi, G., Su, Z., Glondu, S., „Deckard: Scalable and accurate tree-based detection of code clones“. in: *Proceedings of the 29th international conference on Software Engineering*, 2007.

Kamiya, T., Kusumoto, S., Inoue, K., „CCFinder: a multilinguistic token-based code clone detection system for large scale source code“, in: *IEEE Transactions on Software Engineering*, 2002, S. 654–670.

Kapser, C. J., Godfrey, M. W., „“Cloning considered harmful” considered harmful: patterns of cloning in software“, in: *Empirical Software Engineering*, 2008, S. 645.

Knuth, D. E., Pardo, L. T., „The early development of programming languages“, in: *A history of computing in the twentieth century*, 1980, S. 197–273.

Krinke, J., „Identifying similar code with program dependence graphs“. in: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001.

Kumar, B., Singh, S., „Code clone detection and Analysis using Software Metrics and Neural Network-A Literature Review“, in: *Complexity*, 2015, S. 3.

Kumar, D. R., Chanakya, G., „Refactoring Framework for Instance Code Smell Detection“, in: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 2014,

Lämmel, R., Jones, S. P., „Scrap your boilerplate: a practical design pattern for generic programming“, in: *ACM SIGPLAN Notices*, 2003, ??

Liggesmeyer, P., *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Berlin, 2002.

Martin, R. C., *Clean code: a handbook of agile software craftsmanship*. Pearson Education, Westford, 2009.

Mazinanian, D., Tsantalis, N., Stein, R., Valenta, Z., „JDeodorant: clone refactoring“. in: *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016.

McCabe, T. J., „A complexity measure“, in: *IEEE Transactions on software Engineering*, 1976, S. 308–320.

Oracle (2017), *Using Enhanced For-Loops with Your Classes*, URL: `https://blogs.oracle.com/corejavatechtips/using-enhanced-for-loops-with-your-classes` (besucht am 25. 10. 2017).

o.V. (2017), *Wookieepedia - the star wars wiki*, URL: `http://starwars.wikia.com/wiki/Cloning/Legends` (besucht am 24. 07. 2017).

Rattan, D., Bhatia, R., Singh, M., „Software clone detection: A systematic review“, in: *Information and Software Technology*, 2013, S. 1165–1199.

Roy, C. K., Cordy, J. R., „Scenario-based comparison of clone detection techniques“. in: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, 2008.

Roy, C. K., Cordy, J. R., Koschke, R., „Comparison and evaluation of code clone detection techniques and tools: A qualitative approach“, in: *Science of computer programming*, 2009, S. 470–495.

Singh, G., Singh, D., Singh, V., „A study of software metrics“, in: *IJCEM International Journal of Computational Engineering & Management*, 2011, S. 22–27.

Smith, R., Horwitz, S., „Detecting and measuring similarity in code clones“. in: *Proceedings of the International workshop on Software Clones (IWSC)*, 2009.

Speicher, D., Bremm, A., „Clone removal in java programs as a process of stepwise unification“, in: *arXiv preprint arXiv:1301.2447*, 2013,

Svajlenko, J., Roy, C. K., „Evaluating modern clone detection tools“. in: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 2014.

Toegl, R., Winkler, T., Nauman, M., Hong, T. W., „Specification and standardization of a java trusted computing api“, in: *Software: Practice and Experience*, 2012, S. 945–965.

Toomim, M., Begel, A., Graham, S. L., „Managing duplicated code with linked editing“. in: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, 2004.

Ullenboom, C., *Java ist auch eine Insel*. Galileo Computing, Bonn, 2006.

Urma, R.-G., Fusco, M., Mycroft, A., *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Manning Publications, Shelter Island, 2014.

Warburton, R., *Java 8 Lambdas: Pragmatic Functional Programming*. O'Reilly, Sebastopol, 2014.

Watson, R., Stamnes, M., Jeannot-Schroeder, J., Spyridakis, J. H., „API documentation and software community values: a survey of open-source API documentation“. in: *Proceedings of the 31st ACM international conference on Design of communication*, 2013.

Weckerle, V., „CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection“, in: *Master's thesis, Freie Universität Berlin, Germany*, 2008,

Ehrenwörtliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

Leipzig, 9. April 2018